

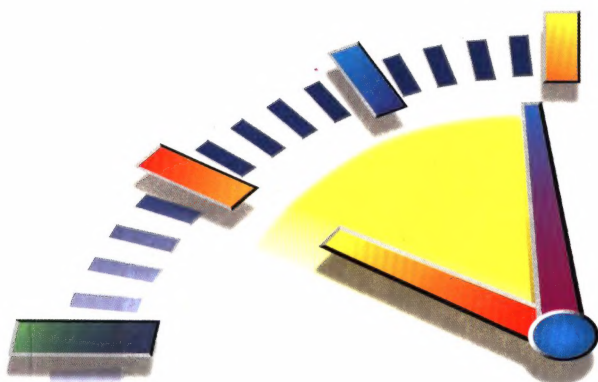
SAMS
Освой самостоятельно!

C++

**ВТОРОЕ
ИЗДАНИЕ**

**Короткие уроки —
быстрые
результаты**

*Освойте очередное средство языка
C++ всего за 10 минут*



SAMS

Джесс Либерти

10 минут на урок

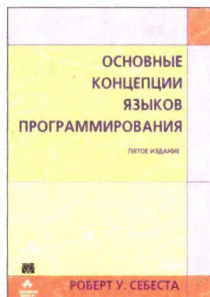
Базовые знания программиста



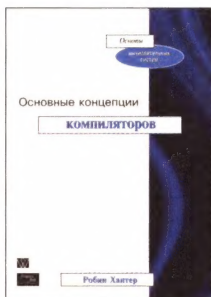
ISBN 5-8459-0498-6



ISBN 5-8459-0080-8



ISBN 5-8459-0192-8



ISBN 5-8459-0360-2



ISBN 5-8459-0261-4



ISBN 5-8459-0189-8



ISBN 5-8459-0330-0



ISBN 5-8459-0579-6



ISBN 5-8459-0179-0

... и много других книг Вы найдете на наших сайтах



www.dialektika.com



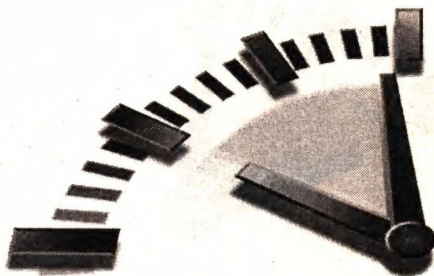
www.williamspublishing.com



www.ciscopress.ru

SAMS
Освой самостоятельно

C++



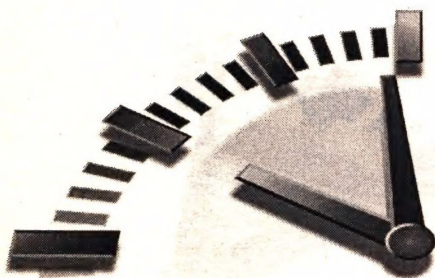
Джесс Либерти

10 минут на урок

ВТОРОЕ ИЗДАНИЕ

SAMS
Teach Yourself

C++



Jesse Liberty

in 10 minutes

SECOND EDITION

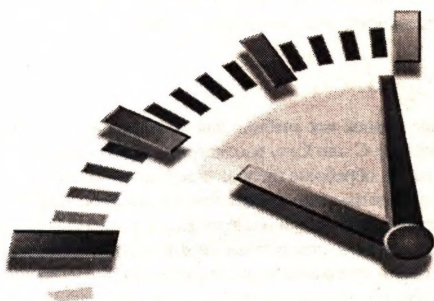
SAMS

800 East 96th St., Indianapolis, Indiana, 46290 USA



SAMS
Освой самостоятельно

C++



Джесс Либерти

10 минут на урок

ВТОРОЕ ИЗДАНИЕ



Москва • Санкт-Петербург • Киев
Издательский дом "Вильямс"
2004

ББК 32.973.26-018.2.75

Л55

УДК 681.3.07

Издательский дом "Вильямс"

Главный редактор *С.Н. Тригуб*

Зав. редакцией *В.В. Александров*

Перевод с английского и редакция *Я.К. Шмидского*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, <http://www.williamspublishing.com>

Либерти, Джесс.

Л55 Освой самостоятельно C++. 10 минут на урок, 2-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 352 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0621-0 (рус.)

В данной книге, задуманной как учебник для начинающих и как пособие для тех, кто уже знаком с языком C или C++, рассмотрены все важные средства языка C++: операторы и операции, обработка ошибок и исключений, управляющие конструкции, данные, управление памятью, раздельная компиляция, указатели, ссылки, ввод-вывод, классы, перегрузка функций и операторов, полиморфизм классов, объявление и определение шаблонов, стандартная библиотека C++. Особое внимание в книге уделяется технологии программирования на языке C++. Подробно рассмотрены все этапы разработки и сопровождения программ. Теоретические положения демонстрируются на примере построения программы калькулятора. Книга написана доступным, простым языком. Она будет полезна не только начинающим, но и тем, кто уже принимал участие в разработке больших программных проектов.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Sams Publishing.

Authorized translation from the English language edition published by Sams Publishing Copyright © 2002

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2004

ISBN 5-8459-0621-0 (рус.)

ISBN 0-672-32425-3 (англ.)

© Издательский дом "Вильямс", 2004

© Sams Publishing, 2002

Оглавление

Введение	20
Урок 1. Начало	23
Урок 2. Вывод на пульт — стандартный вывод	33
Урок 3. Вычисления	39
Урок 4. Ввод чисел	51
Урок 5. Условные операторы <code>if</code> и принятие решений в программах	57
Урок 6. Обработка исключений	63
Урок 7. Функции	67
Урок 8. Разделение кода на модули	79
Урок 9. Циклы: <code>do</code> и <code>while</code>	89
Урок 10. Вложенные циклы и сложные логические выражения	97
Урок 11. Переключатели (инструкции выбора <code>switch</code>), статические переменные и ошибки во время выполнения	105
Урок 12. Массивы, циклы, операторы приращения и декремента	113
Урок 13. Память: динамическая память, стеки и указатели	121
Урок 14. Испытание, или тестирование	135
Урок 15. Структуры и типы	149
Урок 16. Файловый ввод-вывод	167
Урок 17. Классы: структуры с функциями	177
Урок 18. Улучшение программы, или рефакторинг, — переразложение калькулятора на классы	189

Урок 19. Реализация калькулятора как системы классов	195
Урок 20. Остальные классы калькулятора	213
Урок 21. Перегрузка функций и операторов	227
Урок 22. Наследование	243
Урок 23. Испытание объектов с помощью наследования	261
Урок 24. Абстрактные классы, множественное наследование и статические члены	269
Урок 25. Шаблоны	289
Урок 26. Эффективность: оптимизация в C++	307
Урок 27. Итоги, или повторение пройденного	315
Приложение А. Операторы	321
Приложение Б. Старшинство операторов	337
Предметный указатель	339

Содержание

Введение	20
Урок 1. Начало	23
Цель	23
Язык C++	24
Подготовка к программированию	25
C++, ANSI C++, Windows и другие часто путаемые вещи	25
Компилятор и редактор	26
Начинаем новый проект	27
Цикл разработки	27
Усовершенствование программы	29
Простая программа	29
Части программы	30
Ошибки, обнаруживаемые во время компиляции	31
Резюме	32
Урок 2. Вывод на пульт — стандартный вывод	33
Расширение пустой программы	33
Включение файлов символ за символом: оператор #include	35
Пространства имен	35
Комментарии	36
Пробельные символы	37
Функции	37
Инструкция cout: вывод слов	38
Резюме	38
Урок 3. Вычисления	39
Выполнение вычислений и отображение их результатов	39
Выражения	39

Порядок вычисления	40
Вложенные круглые скобки	41
Выражения в <code>cout</code>	41
Использование входного потока	41
Переменные	43
Размер памяти	44
Использование переменных и констант типа <code>int</code>	45
Типы переменных и допустимые названия (имена)	46
Типы нецелочисленных переменных	47
Строки	48
Чувствительность к регистру	48
Ключевые слова	48
Резюме	49
Урок 4. Ввод чисел	51
Ввод чисел	51
В чем ошибка?	52
Что же происходит не так, как надо?	54
Резюме	56
Урок 5. Условные операторы <code>if</code> и принятие решений в программах	57
Обработка ошибок во входном потоке	57
Инструкция <code>if</code> (если)	58
Как принимать решение?	59
Восстановление после ошибки	60
Резюме	62
Урок 6. Обработка исключений	63
Обработка исключений — более лучший способ	63
Зачем использовать исключения?	66
Резюме	66
Урок 7. Функции	67
Что такое функция?	67
Определение функций	68
Разбиение программы примера на несколько функций	70

Функция без аргументов и возвращаемых значений	72
Функция без аргументов, но с локальными переменными и возвращаемым значением	72
Функция с аргументами и возвращаемым значением	73
Вызов функции с аргументами, которые сами являются вызовами функций	74
Улучшение кода, или переразложение на классы	74
Где следует помещать код функций?	75
Глобальные переменные	77
Испытание	78
Резюме	78
Урок 8. Разделение кода на модули	79
Что такое модуль?	79
Зачем использовать модули?	80
Что находится в заголовочном файле?	80
Создание заголовочного файла	80
Как выглядит файл реализации?	81
Изменение названий при создании библиотеки	83
Вызов функций	85
Раздельная компиляция	87
Испытание	87
Резюме	88
Урок 9. Циклы: do и while	89
Что же у нас получилось и как этим воспользоваться?	89
Повторение выполнения	89
Выполнение инструкций по крайней мере один раз	90
Условие цикла	91
Размещение инструкций try и catch	93
Выполняем ноль или большее количество раз	94
Резюме	96

Урок 10. Вложенные циклы и сложные логические выражения	97
Вложение циклов	97
Операторы сравнения	99
Упрощение с помощью логических переменных	102
Резюме	104
Урок 11. Переключатели (инструкции выбора switch), статические переменные и ошибки во время выполнения	105
Переключатели: инструкции выбора switch	105
Обобщение калькулятора	107
Переключатель (инструкция выбора switch)	109
Статические локальные переменные	110
Самостоятельный вызов исключения	111
Обработка нового исключения	111
Резюме	112
Урок 12. Массивы, циклы, операторы приращения и декремента	113
Использование массива для создания ленты калькулятора	113
Лента Tape	114
Цикл for	116
Запись после конца массива	117
Увеличение и уменьшение	118
Лента калькулятора в сумматоре	119
Резюме	120
Урок 13. Память: динамическая память, стеки и указатели	121
Динамическая память и стек	121
Указатели, ссылки и массивы	124
Массивы на самом деле представляют собой указатели	126
Массивы для ленты Tape () в динамической памяти	127

Ссылки	132
Указатели опасны	133
Удаление из динамической памяти	133
Удаление массивов	134
Резюме	134
Урок 14. Испытание, или тестирование	135
Почему критичны испытания программ, использующих динамическую память?	135
Обобщение калькулятора с помощью “небольшого языка”	136
Изменения в главной функции <code>main()</code>	137
Изменения в сумматоре <code>Accumulator()</code>	139
Изменения в функциях ввода	141
Функция самотестирования <code>SelfTest</code>	141
Функция <code>TestOK()</code>	142
Небольшое изменение в ленте <code>Tape()</code>	143
Выполнение программы	144
Сокращение вычислений при вычислении операндов логического оператора <code>&&</code>	144
Что было неправильно?	145
Устранение ошибки и повторный запуск	145
Отладка без отладчика	145
Как поймать волка методом деления пополам (дихотомии) его территории обитания	146
Печать значений	146
Включение и отключение отладки с помощью команд <code>#define</code>	146
Резюме	147
Урок 15. Структуры и типы	149
Организация разработки программ	149
Объявление перечислимых типов	151
Использование перечислений в сумматоре <code>Accumulator()</code>	152
Объявление структурных типов	155
Структуры в стеке	155
Структуры в динамической памяти	156

Однонаправленный связный список со структурами для ленты	157
Указатели на функции и обратные вызовы	159
Вызов функции с помощью указателя	163
Резюме	165
Урок 16. Файловый ввод-вывод	167
Сохранение ленты между сессиями	167
Файловые потоки <code>fstream</code>	167
Запись ленты на диск в потоке	169
Открытие и закрытие файла ленты	169
Как заставить работать <code>StreamTape()</code>	171
Указание имени файла для ленты	172
Выбор ленты	172
Проигрываем ленту, чтобы восстановить состояние	172
Резюме	176
Урок 17. Классы: структуры с функциями	177
Класс как мини-программа	177
Классы и их экземпляры	178
Объявление класса	178
Заголовок и реализация	180
Вызов функций-членов класса	180
Типы, вложенные в классы	181
Конструкторы и деструкторы	182
Раздел инициализации в конструкторе	183
Тело конструктора	183
Конструкторы с параметрами	184
Множественные конструкторы	184
Деструкторы	185
Конструктор копирования и его назначение	186
Ослабление правила “Объявить перед использованием” в классах	188
Резюме	188

Урок 18. Улучшение программы, или рефакторинг, — переразложение калькулятора на классы	189
Перенос функций в классы	189
Диаграмма UML	189
Диаграмма UML для калькулятора	190
Отличия	194
Резюме	194
Урок 19. Реализация калькулятора как системы классов	195
Система обозначений класса	195
Частные и общедоступные члены aRequest	197
Инициализация	198
Внутреннее состояние	200
Соглашения об именовании	203
Перемещение кода функций в член-функции	205
Объект как структура обратного вызова	211
Кто выделяет память, кто удаляет, кто использует и что разделяется (используется совместно)	211
Резюме	212
Урок 20. Остальные классы калькулятора	213
Использование классов Стандартной библиотеки C++	213
Использование библиотеки классов в aTape	214
Интерфейс пользователя в объекте main.cpp	219
Резюме	225
Урок 21. Перегрузка функций и операторов	227
Объявление перегруженных членов-функций в классе	227
Перегруженные конструкторы	232
Что означает перегрузить оператор?	233
Перегрузка оператора может быть опасна	234

Перегрузка оператора <<	234
Ключевое слово const (константа)	
и перегруженные операторы	238
Перегрузка: ключевые положения	239
Перегрузка присваиваний и конструктора копии	240
Резюме	241
Урок 22. Наследование	243
Объявление наследования	243
Новые и измененные классы	244
Создание производного класса ленты Tape	246
Реализация производного класса	247
Ссылки на объект своего класса и суперкласса	249
Уровни наследования	251
Переопределение функций	252
Защищенный доступ	254
Что означает ключевое слово	
virtual (виртуальный)?	255
Виртуальные конструкторы и деструкторы	258
Виртуальные функции-члены	258
Вызов суперкласса	259
Резюме	260
Урок 23. Испытание объектов с помощью наследования	261
Написание инструментальных средств тестирования	261
Испытание классов с помощью заранее подготовленных тестов	262
Регрессивные испытания	265
Запись в файлы входных и выходных данных для испытаний	265
Использование наследования	266
Резюме	267

Урок 24. Абстрактные классы, множественное наследование и статические члены	269
Создание интерфейсов	269
Чистые виртуальные функции	270
Объявление абстрактного класса	271
Реализация абстрактного класса	273
Изменения в <code>aController</code>	274
Фабрика объектов	276
Абстрактные классы в дереве наследований	277
Множественное наследование	279
Статические поля и функции в классах	282
Статические (<code>static</code>) члены класса	283
Увеличение (инкрементирование) и уменьшение (декрементирование) счетчика экземпляров	285
Вывод главной программы <code>main.cpp</code>	286
Резюме	288
Урок 25. Шаблоны	289
Сила и слабость шаблонов	289
Объявление и использование шаблонов	290
Калькулятор как система шаблонов	293
Изменение <code>anAccumulator</code>	
и <code>aBasicAccumulator</code>	296
Использование шаблонов	302
Выполнение испытания	304
Несколько замечаний о шаблонах	304
Сила и слабость шаблонов	305
Резюме	306
Урок 26. Эффективность: оптимизация в C++	307
Выполняем быстрее и уменьшаем объем	307
Встроенный код	308
Приращение (увеличение) и уменьшение	310
Шаблоны или универсальные классы?	311
Хронометраж кода	311
Размер программы и структуры данных	312
Резюме	313

Урок 27. Итоги, или повторение пройденного	315
Как усовершенствовать калькулятор?	315
Добавление возможностей отмены ввода (Undo) и повторения ввода (Redo)	315
Добавление поименованных переменных	316
Использование калькулятора в более мощной программе, например в электронной таблице	316
Использование графического интерфейса пользователя (Graphical User Interface, GUI)	316
Изученные уроки	316
Думайте о классах и объектах	317
Развитие программы	317
Частое улучшение кода (переразложение на классы)	318
Поддержка контракта	318
Частые испытания	319
Подумайте об эффективности в подходящее время	319
Не усложняйте простых вещей...	319
Соблюдайте соглашения об именовании	319
Будьте терпеливы, работая с компилятором	320
Приложение А. Операторы	321
Приложение Б. Старшинство операторов	337
Предметный указатель	339

Об авторах

Джесс Либерти — автор более дюжины книг по программированию, включая и такие заслужившие международное признание бестселлеры, как *Teach Yourself C++ in 21 Days*, выпущенный издательством Sams (есть перевод: *Освой самостоятельно C++ за 21 день*, выпущенный Издательским домом “Вильямс”) и *Programming C#* (Программирование на C#) (издательство O'Reilly). Джесс — президент компании Liberty Associates, Inc. (<http://www.LibertyAssociates.com>), где он обучает работе на новой платформе .NET, программированию и проводит консультации. Ранее он был вице-президентом крупного банка Citibank, ведущим инженером по программному обеспечению и архитектором программного обеспечения в таких известных компаниях, как AT&T, Ziff Davis, PBS (государственная служба радиовещания) и Xerox.

Марк Кашман — разносторонний профессионал в области информационных технологий, который занимал должности от архитектора программного обеспечения до главы службы информационных технологий. Его недавним увлечением были деловые Internet-приложения (eBusiness), где он был ответственным за разработку технологий уровня предприятия, а также принятие и интеграцию разрабатываемых каркасов на уровне предприятия (EAI — enterprise application integration), и архитектуру Web-презентаций на J2EE. Он — член Borland TeamB, поддерживающей C++Builder, автор разделов C++Builder 5 Developer's Guide (Sams) и многих статей в журнале *C++Builder Developer's Journal*. На его сайте, называемом Temporal Doorway (<http://www.temporaldoorway.com>), размещаются обучающие программы по программированию на C++ и Java. В настоящее время он разрабатывает обзор маршрутов Новой Англии (<http://www.newenglandtrailreview.com>), т.е. сайт, с которого можно будет получить доступ к обширной базе данных маршрутов пешего туризма в Новой Англии.

Посвящение

Эта книга посвящается людям, работающим ради нашей безопасности и свободы, а также памяти об 11 сентября.

Джеесс Либерти

Благодарности

Прежде всего, я должен поблагодарить мое семейство, которое продолжает поддерживать меня в моих авторских трудах и терпит мой безумный распорядок. Я также хочу поблагодарить сотрудников издательства Sams, особенно Кэрол Аккерман (Carol Ackerman), Мэтт Перселл (Matt Purcell) и Мэтт Винолда (Matt Wynalda). Мои благодарности Марку Кашману (Mark Cashman) и Кристоферу Макги (Christopher McGee) за то, что помогли сделать это издание лучше, чем когда-либо.

Джесс Либерти

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: <http://www.williamspublishing.com>

Информация для писем из:

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

Введение

Если C++ заинтересовал вас, но у вас никогда не хватало времени, чтобы подробно изучить его особенности и возможности...

Если ваши навыки в программировании на C++ немного притупились, потому что вы были заняты другой работой...

Если вы хотите изучить программирование на C++, но не хотите покупать книгу более тяжелую, чем ваш компьютер...

Если вы хотите что-то узнать, но не хотите пробираться через джунгли справочников, чтобы найти нужные сведения...

Освой самостоятельно C++. 10 минут на урок — именно та книга, которая вам нужна.

Учитесь быстро

Конечно, вы не можете тратить многие часы на чтение. Вы просто хотите знать язык и получить практические советы по применению C++. Поэтому в книге принят подход, который позволит вам увидеть, как создаются и разрабатываются реальные программы на C++, программы, которые дают реальные результаты.

Это руководство — не всеобъемлющий манускрипт по C++, составленный из огромных глав, переписанных из справочного описания. В нем внимание сосредоточено на наиболее важных аспектах языка, его основах и более сложных вопросах, причем материал книги представлен в виде отдельных уроков, которые организованы так, чтобы на каждый из них тратилось примерно 10 минут.

Еще один подход

Если вы хотите научиться писать программы, но никогда не делали этого прежде, вам потребуется помощь по всем вопросам программирования. Если же вы профессиональный программист, вам нужно знать, какую роль играет C++ на всех стадиях жизненного цикла программы.

Если что-нибудь из перечисленного выше подходит вам, эта книга — для вас.

В книге прослежена разработка одной единственной программы, от начала до конца — это позволит вам сконцентрироваться на языке, а не на новом приложении для каждого примера. Вы научитесь создавать, расширять, реструктурировать, проверять и оптимизировать программы на C++, а также устранять ошибки в них. Вы изучите практические советы на реальном примере и тем самым узнаете, как заставить C++ выполнить ваше задание.

Что вы узнаете о C++?

Эта книга содержит все, что нужно знать о C++, включая его основы и более сложные особенности, причем по всем вопросам даются понятные объяснения и приводятся диаграммы, помогающие вам извлечь наибольшую пользу из каждого десятиминутного урока:

- арифметика;
- переменные и константы;
- управляющие инструкции (конструкция если (if) и переключатель switch) и логические выражения;
- циклы (do, while и for);
- функции;
- ввод и вывод информации от пользователей или из файлов;
- обработка ошибок и исключений;
- отдельная трансляция;
- массивы, указатели и ссылки;
- указатели на функции;
- получение памяти из динамически распределяемой области памяти;
- структуры данных и определяемые пользователем типы;
- классы и элементы класса;
- функции и перегрузка операторов;
- наследование и множественное наследование;
- полиморфизм класса;
- шаблоны.

Когда вы закончите читать эту книгу, независимо от того, новичок вы или профессиональный программист, вы сможете создавать и поддерживать программы на уровне профессионала в C++.

Соглашения, используемые в этой книге

Каждый урок этой книги посвящен отдельному аспекту программирования на C++. Следующие пиктограммы облегчают чтение книги.



Эта пиктограмма отмечает места, в которых обсуждаются области программирования, где новички сталкиваются с неприятностями. В тексте предлагаются практические решения проблем.



Эта пиктограмма указывает, что здесь простым языком разъясняются концепции и процедуры.



Эта пиктограмма указывает, что здесь обсуждаются важные идеи, которые упрощают программирование и позволяют избежать беспорядка.



Эта пиктограмма указывает, что здесь вводятся (или определяются) новые или незнакомые термины простым русским языком.

УРОК 1

Начало



В этом уроке вы узнаете, как подготовиться к написанию программы на C++, как ее спроектировать, создать и изменить.

Цель

В этой книге прослеживается жизненный цикл конкретной программы — от момента ее создания до реальной эксплуатации. Подобно многим программам, вначале ее концепция очень проста, но затем она усложняется с каждым уроком, чтобы предложить пользователю все большее количество возможностей.

Цель этого подхода состоит в том, чтобы вы могли сосредоточиться на языке и на его применении. Работая в основном с единственным примером на протяжении всей книги, вы сможете сконцентрироваться на новых особенностях и их поддержке в языке C++. Всего лишь несколько частей нашей программы-примера созданы исключительно для того, чтобы продемонстрировать особенности языка. Большинство добавлений и изменений обусловлены реальными требованиями к программе и вытекают из целей ее разработки — они, несомненно, понадобились бы в будущих версиях программы.

В этой книге вы изучите:

- язык C++;
- жизненный цикл программных разработок;
- эволюционный или адаптивный метод разработки, в соответствии с которым сначала разрабатывается простая программа, которая затем постепенно усложняется. Этот метод разработки часто используется в профессиональном программировании.

Язык C++

Язык C++ был создан как усовершенствованная версия языка C. C был создан Брайаном Керниганом (Brian Kernighan) и Деннисом Ритчи (Dennis Ritchie) в Bell Labs в период с 1969 по 1973 год. Он первоначально был предназначен для программирования служебных программ нижнего уровня типа операционных систем (Керниган и Ритчи разрабатывали Unix). Предполагалось, что он заменит ассемблер. Программы, написанные на ассемблере, было очень трудно читать. Кроме того, на ассемблере очень трудно создать программы, которые можно было бы легко разделить на отдельные модули. C получил широкое распространение и стал ключевым языком для Unix и в конечном счете для Windows.

C самого начала создавался для написания высокоэффективных программ, и эта же цель преследовалась и при создании C++.

При создании программ на C приходится применять *процедурный* стиль программирования. Процедурное программирование позволяет создавать программы, которые являются набором функций или процедур, обрабатывающих данные, чтобы получить определенный результат. Функции могут обращаться к другим функциям для получения услуг и помощи, что позволяет применить стратегию “разделяй и властвуй” и тем самым упростить решение задач.

Кроме того, C — язык *со строгим контролем типов*. Это означает, что каждый элемент данных в C имеет тип и может использоваться с другими частями данных только так, как определено в их типах. Языки *со слабым контролем типов* (такие как BASIC) либо игнорируют, либо скрывают этот важный принцип. Строгий контроль типов гарантирует, что программа правильна, точнее, выполняет осмысленные действия еще до первого ее запуска.

Бьярн Страуструп (Bjarne Stroustrup) разработал язык C++ в 1983 году как расширение языка C. C++ имеет большинство возможностей C. Фактически многие программы на C++ выглядят точно так же, как и на C. В первой части этой книги вы будете разрабатывать именно такие программы. Вы можете посещать сайт Страуструпа <http://www.research.att.com/~bs/C++.html>. Это — превосходный источник дополнительного материала.

В C++ предусмотрен *объектно-ориентированный* стиль программирования. Когда вы начнете писать объектно-ориентированные программы, вы увидите аналогии и различия между процедурным и объектно-ориентированным программированием.

С точки зрения объектно-ориентированного программирования, программа представляет собой набор классов, которые используются для генерации объектов. Каждый класс содержит данные и функции. Объект может обращаться к объектам других классов за услугами и помощью. Поскольку данные скрыты глубоко внутри класса, объектно-ориентированные программы более безопасны и их легче изменять, чем процедурные программы, в которых изменение структур данных может затрагивать все функции программы.

Классы имеют в качестве своих элементов (членов) данные и функции, а так как члены-данные и члены-функции очень похожи на аналоги в процедурном программировании, то именно с них мы и начнем.

Подготовка к программированию

Первый вопрос, который нужно задать при подготовке к проектированию любой программы: в чем состоит проблема, которую я пытаюсь решить? Каждая программа должна иметь ясную, хорошо сформулированную цель, и вы увидите, что даже самая простая версия программы в этой книге будет ее иметь.

C++, ANSI C++, Windows и другие часто путаемые вещи

В книге *Освой самостоятельно C++*. 10 минут на урок не делается никаких предположений о вашем компьютере. В этой книге рассматривается Стандартный C++ ISO/ANSI (который называется просто Стандартным C++). Международная организация по стандартизации (International Organization for Standardization — ISO), в которую входит Американский национальный институт стандартов (American National Standards

Institute — ANSI), устанавливает стандарты, а также издает документы, в которых точно описывается, как должны выглядеть и работать правильные программы на C++. Вы должны научиться создавать такие программы на любой системе.

Вы ничего не найдете в этой книге об окнах, списках, графике и т.д. Наборы классов и функций (часто называемые *библиотеками*), взаимодействующие непосредственно с операционной системой (например Windows, Unix или Mac), обеспечивают эти специальные возможности, но на них не распространяется стандарт ISO/ANSI. Таким образом, в программах этой книги используется консольный ввод-вывод, который более прост и доступен на каждой системе.

Программу, которую вы создадите, можно легко приспособить и для использования возможностей графического интерфейса пользователя (graphical user interface — GUI), так что вы сможете воспользоваться почерпнутыми из этой книги знаниями, чтобы работать с нужными вам библиотеками.

Компилятор и редактор

Компилятор — это программа, которая читает программу, написанную на читаемом человеком языке (*исходный текст*) и преобразовывает ее в файл (*выполнимую* программу), который может быть выполнен на компьютере операционной системой (например Windows, Unix или Mac). *Редактор* — программа (такая как знакомый вам Блокнот Windows), которая позволяет напечатать исходный текст программы и сохранить его в файле. Чтобы читать эту книгу, конечно же, нужен, по крайней мере, один компилятор и один редактор.

В этой книге предполагается, что вы умеете с помощью редактора создавать, сохранять и изменять текстовые файлы и что вы знаете, как использовать ваш компилятор и любые другие необходимые инструментальные средства типа компоновщика. Все необходимые сведения по этим вопросам содержатся в справочной системе операционной системы и документации к компилятору.

Коды, представленные в этой книге, были откомпилированы компилятором Borland C++Builder 5 в строгом режиме ANSI.

Они также были проверены на Microsoft Visual Studio версии 6. Вам доступны многочисленные свободно распространяемое обеспечение и компиляторы общего пользования (shareware), включая один от Borland (<http://www.Borland.com>) и известный компилятор gcc (<http://gcc.gnu.org>). Вы можете найти информацию о доступных свободно распространяемых компиляторах и компиляторах общего пользования (shareware) для C++ на Web-странице этой книги <http://www.sampublishing.com>.

Начинаем новый проект

Для каждой части создаваемой программы нужно сделать каталог. Все необходимые сведения должны быть в документации к вашей системе. Рекомендуется создать каталог верхнего уровня для всех программ из этой книги, а затем ниже по одному каталогу для примера каждого урока.

Лучше всего с помощью редактора создать файлы с исходным текстом на C++ и сохранять их в каталоге для соответствующего урока. Обычно исходные файлы имеют расширение файла .cpp в конце, что и идентифицирует их как код на C++.

Если вы используете *интегрированную среду разработки* (Integrated Development Environment — IDE), например Borland C++Builder или Visual C++ Microsoft, обычно лучше всего создать новый проект, выбрав File⇒New. В таких средах программу нужно создавать на основе проекта с консольным вводом-выводом (console-type projects). Каждый проект лучше всего сохранять в отдельном каталоге для каждого урока, и там же нужно хранить все исходные файлы проекта. То, как это делается, должно быть описано в документации к интегрированной среде разработки.

Цикл разработки

Если бы каждая программа работала с первого раза, весь цикл разработки состоял бы из написания программы, компиляции исходного текста и его выполнения. К сожалению, почти каждая программа, даже самая простая, содержит ошибки. Некоторые ошибки являются синтаксическими и выявляются на этапе трансляции, но некоторые обнаруживаются только при выполнении программы.

Фактически, разработка каждой программы проходит через следующие стадии.

- **Анализ** — решите, что программа должна делать.
- **Проектирование** — определите, как программа будет делать то, что требуется делать.
- **Редактирование** — создание исходного текста на основе проекта.
- **Компиляция** — используйте компилятор, чтобы превратить программу в файл, который может выполнить компьютер. Компилятор сгенерирует сообщения об ошибках, если инструкции на C++ написаны неправильно. Эти часто загадочные сообщения об ошибках нужно понять — и устранить все ошибки в коде, пока не добьетесь “безошибочной” компиляции.
- **Компоновка, или редактирование связей** — обычно компилятор автоматически свяжет безошибочный код с любыми нужными ему библиотеками.
- **Испытание, или тестирование** — компилятор не ловит каждую ошибку, так что программу приходится выполнять, причем иногда со специально запланированными входными данными, и убедиться, что она не делает что-либо не так во время выполнения. Некоторые ошибки во время выполнения приводят к тому, что операционная система останавливает программу, но в других случаях программа выдает неправильные результаты.
- **Отладка** — из-за ошибок, обнаруживаемых во время выполнения программы, приходится потрудиться над программой, чтобы найти, что же в ней не так. Проблема иногда заключается в недостатках проекта, иногда — в неправильном использовании особенностей языка, а иногда — и в неправильном использовании операционной системы. Отладчики — это специальные программы, которые помогают выявить такие проблемы. Если отладчика нет, в исходный текст приходится вставлять код, который сообщает, что программа делает на очередном этапе выполнения.

Независимо от типа найденной ошибки, ее необходимо устранить, для этого придется отредактировать исходный текст, перетранслировать его, пересобрать, а затем выполнить повторный запуск программы. И все это придется делать до тех пор, пока не исправите программу полностью. Вы научитесь всему этому в процессе чтения этой книги.

Усовершенствование программы

Как только вы заканчиваете программу, почти всегда оказывается, что в ней нужно сделать кое-что дополнительно или же не так, как было запланировано. Пользователям потребуется новая возможность, или они найдут ошибку во время эксплуатации программы, которую вы не обнаружили при испытании. Или же вам не понравится внутренняя структура программы и захочется улучшить ее, чтобы облегчить чтение и сопровождение программы.

Простая программа

Приведенная ниже простая программа фактически ничего не делает, но компилятор об этом даже не догадывается. Вы можете скомпилировать и выполнить эту программу.



Номера строк в коде

Приведенный ниже листинг содержит номера строк. Эти числа (номера) предназначены для ссылок в тексте книги. Их ни в коем случае не нужно вводить в редакторе. Например, в строке 1 из листинга 1.1 вы должны ввести

```
int main(int argc, char* argv[])
```

Листинг 1.1. main.cpp — пустая программа

```
1: int main(int argc, char* argv[])
2: {
3:     return 0;
4: }
```

Убедитесь, что вы ввели программу точно так, как показано в листинге (за исключением номеров строк). Уделите должное внимание знакам пунктуации. В конце 3-й строки не забудьте поставить точку с запятой!

В C++ каждый символ, включая и символы пунктуации, имеет определенное значение и потому должен стоять на своем месте. Кроме того, в C++ имеет значение и регистр, например, `return` и `Return` — это разные слова.

Части программы

Эта программа состоит из одной функции, названной `main` (главная). Эта функция, начинающаяся в строке 1, имеет два параметра (они указаны внутри круглых скобок) и возвращает в качестве результата число (на это указывает первое ключевое слово `int`).

Функция — отдельная группа строк в тексте программы, предназначенная для решения определенной задачи. Функция начинается с заголовка, причем имя функции — это второе слово. После открывающейся фигурной скобки (`{`) следует тело, которое заканчивается закрывающей фигурной скобкой (`}`). После закрывающей фигурной скобки при желании можно поставить точку с запятой. Подробнее функции будут обсуждаться в уроке 7, “Функции”.

Функция `main` (главная) необходима во всех программах на C++. Параметры (часто называемые также аргументами) эта функция получает от системы. Вот эти параметры:

- `int argc` — счетчик слов в строке, напечатанной для запуска программы;
- `char* argv[]` — строка, напечатанная для запуска программы, притом разбитая на слова.

Функция имеет *заголовок* (строка 1) и *тело* (строки 2-4). *Фигурные скобки* в строках 2 и 4 показывают, где начинается и кончается тело. Любая последовательность строк, начинающаяся с открывающей фигурной скобки и заканчивающаяся закрывающей фигурной скобкой, называется *блоком*, или *составной инструкцией*. Строка 3 — *простая инструкция*, которая возвращает системе число, когда программа заканчивается — в нашей программе возвращается 0.

Эта программа представляет собой единственный файл с расширением `.crr`. Такой файл называется также *модулем*. Иногда модуль состоит из двух файлов — заголовочного файла (его имя заканчивается на `.h`) и `.crr`-файла. Файлу главной функции — `main.crr` — заголовок не нужен и потому он никогда не имеет его.



Возвращаемое значение

Возвращаемое значение возвращается всегда, но очень часто не используется (на Unix и DOS оно иногда проверяется в пакетных файлах, чтобы сообщить об успешном завершении работы или отказе программы).

Ошибки, обнаруживаемые во время компиляции

Ошибки, обнаруживаемые во время компиляции программы, могут быть следствием опечатки или неправильного использования языка. Хорошие компиляторы указывают, что именно вы сделали неправильно и место, где вы сделали ошибку. Иногда они даже указывают, что нужно сделать, чтобы исправить ошибку.



Ошибки пунктуации

Современные компиляторы пытаются найти строку, в которой содержится ошибка, поэтому отсутствие точки с запятой или закрывающей фигурной скобки может “сбить с толку” компилятор, и тогда он может указать на строку, которая сама по себе вполне правильна. Поэтому старайтесь избегать ошибок пунктуации; они могут быть коварными.

Вы можете узнать, как конкретный компилятор реагирует на ошибку, преднамеренно делая ошибки в программе. Если программа `main.crr` выполняется безошибочно, отредактируйте ее: удалите закрывающую фигурную скобку (в строке 4). Тогда получится программа, приведенная в листинге 1.2.

Листинг 1.2. Демонстрация ошибок, обнаруживаемых компилятором

```
1: int main(int argc, char* argv[])
2: {
3:     return 0;
```

Перетранслируйте вашу программу, и вы увидите сообщение об ошибке, которое выглядит примерно так:

```
[C++ Error] Main.cpp(3): E2134 Compound statement missing }
```

```
[ Ошибка C++] Main.cpp (3): E2134 Составная инструкция
                        отсутствие }
```

В этом сообщении об ошибке указан файл, номер строки с ошибкой и указано, в чем состоит проблема.

Иногда сообщение может только приблизительно характеризовать проблему. Например, если в строке 2 вместо первой фигурной скобки поставить закрывающую, т.е. такую, как последняя (в строке 4), то появятся сообщения вроде следующих:

```
[C++ Error] Main.cpp(3): E2141 Declaration syntax error
[C++ Error] Main.cpp(4): E2190 Unexpected }
```

```
[Ошибка C++] Main.cpp(3): E2141 синтаксическая ошибка
                        в объявлении
```

```
[Ошибка C++] Main.cpp(4): E2190 Неожиданная }
```

Иногда одна ошибка вызывает другую, как в последнем случае. Обычно лучше всего исправить несколько первых ошибочных строк и затем программу перетранслировать.

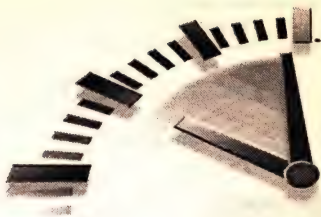
Иногда сообщения об ошибках проще понять, если “думать так, как компилятор”. Компиляторы просматривают исходный текст пословно и по предложениям. Они не понимают смысл программы и ваших намерений.

Резюме

Вы ознакомились с историей создания C++ и узнали, что такое жизненный цикл программы. Вы создали простую программу на C++, откомпилировали ее и научились интерпретировать сообщения об ошибках, генерируемые компилятором.

УРОК 2

Вывод на пульт — стандартный вывод



В этом уроке вы узнаете, как изменить пустую программу так, чтобы она делала кое-что полезное, как работать с библиотеками и как отобразить результаты программы.

Расширение пустой программы

Ваша первая задача состоит в том, чтобы добавить к пустой программе строку, так что программа теперь уже не будет пустой. Эта строка как раз и отобразит результат выполнения программы.

В листинге 2.1 показана новая версия главной программы `main.cpp`. Звездочка перед номером строки указывает, что эта строка является новой в программе. Не вводите *, номер строки и : в код. Эти метки и числа (номера) используются только для ссылок на строки примера программы в описании кода.

Листинг 2.1. Главная программа `main.cpp`, отображающая результат

```
*1: #include <iostream>
*2:
*3: using namespace std; // использовать пространство
                        // имен std
*4:
*5: int main(int argc, char* argv[])
*6: {
*7: // без "using" пришлось бы писать std::cout
```

```
*8:  cout << "Hi there!" << endl; // "endl" = next line =  
                                         // новая строка  
9:      return 0;  
*10: }
```

Вывод

Hi there!

Анализ

Эта программа отображает текст “Hi there!” (“Эй, там!”). Она делает это, используя библиотеку уже написанных компонентов программы, называемую `iostream`. Она так называется потому, что действует таким образом, будто ввод и вывод являются потоком символов (Input and Output Stream of characters).

В строке 1 `#include` компилятору сообщается, что он должен включить заголовочный файл `iostream.h` — компилятор достаточно “хитер” и “знает” все что нужно о файлах с расширением `.h`, так что вам не придется вставлять их самим в программу. Этот заголовочный файл описывает те компоненты библиотеки, которые нужны компилятору для того, чтобы он правильно идентифицировал такие названия (имена), как `cout`.

Включение заголовочного файла для `iostream` позволяет использовать библиотеку `iostream`. Вы можете открыть файл `iostream.h`, если найдете его на вашей системе (обычно подкаталог включаемых файлов находится в каталоге, где установлен компилятор). Впрочем, его исходный текст понять новичку довольно трудно. К счастью, компилятор этого не делает.

Библиотека `iostream` содержит объявление потока стандартного вывода, упомянутого в строке 8 как `cout`. Она также содержит объявление оператора вставки в поток (`<<`) и манипулятора потока (`endl`).

Библиотека `iostream` обычно будет автоматически подключаться к программе на стадии редактирования связей. Этот процесс описан в документации к компилятору.

Результат программы будет отображаться как текст в окне или на экране, в зависимости от того, в какой среде выполняется программа.

Включение файлов символ за символом: оператор `#include`

Первый символ — символ фунта (`#`) — является сигналом для *препроцессора*. Задача препроцессора состоит в том, чтобы читать исходный текст, выискивая строки, которые начинаются с `#`. Когда препроцессор находит их, он изменяет код так, как того требует данная команда. Все это делается еще до того, как компилятор увидит ваш код.

Команда `include` (включить, вставить) указывает, что далее следует имя файла. Указанный файл нужно найти и вставить на место команды. Угловые скобки вокруг имени файла сообщают препроцессору, что поиск нужно выполнять там, где обычно может находиться такой файл. Если компилятор установлен правильно, угловые скобки заставят препроцессор искать файл `iostream.h` в каталоге, который содержит все заголовочные файлы для вашего компилятора.

В результате выполнения команды, записанной в строке 1, в программу должен быть вставлен файл `iostream.h` так, как будто вы напечатали его в программе.

Пространства имен

Почти все в программе имеет название (имя). Когда используются библиотеки, всегда есть шанс, что в одной библиотеке есть нечто, что имеет то же самое название (имя), что и нечто совсем иное в другой библиотеке. Если это случится, компилятор не сможет определить, которую из одноименных вещей вы хотели использовать. Поэтому программисты помещают библиотеки C++ в пространства имен, чтобы можно было сказать, к какому пространству имен принадлежат те или иные названия (имена).

Пространства имен — своего рода контейнеры для названий (имен). Каждое пространство имен само имеет название (имя), которое квалифицирует (уточняет) любые названия (имена) в коде, который их содержит. Например, пространство имен для `iostream` называется `std` (стандартное) и любое

название (имя) из этого пространства имен автоматически уточняется с помощью `std`, например, так: `std::cout`.

Инструкция в строке 3 указывает компилятору, что все названия (имена), для которых пространство имен не указано явно, рассматриваются как принадлежащие пространству имен `std`.

Если эту инструкцию опустить, компилятор сгенерирует следующее сообщение:

```
[C++ Error] Main.cpp(8): E2451 Undefined symbol 'cout'
```

```
[C++ Error] Main.cpp(8): E2451 Undefined symbol 'endl'
```

```
[Ошибка C++] Main.cpp (8): E2451 Неопределенный символ 'cout'
```

```
[Ошибка C++] Main.cpp (8): E2451 Неопределенный символ 'endl'
```

В данном случае сообщение об ошибке компилятор генерирует потому, что не знает, где искать `cout` и `endl`, — из-за того, что мы не указали пространство имен, в котором находятся их определения. Конечно, это можно сделать и без инструкции `namespace`, для этого следует изменить строку 8 так:

```
std::cout << "Hi there!" << std::endl;
```

Однако прочитать это труднее, а поскольку имена из пространства `std` использовались задолго до введения пространств имен в C++, именно инструкция `namespace` помогает сохранить простоту ранее написанных инструкций.

Комментарии

Комментарий — это текст, который объясняет (вам или другим программистам), почему в коде некоторая задача решается именно так, а не как-нибудь иначе. Комментарий не транслируется компилятором в файл исполнимой программы, а служит только для документирования.

В комментариях не обязательно должно объясняться абсолютно все. Однако в них нужно объяснить проект и специфику его реализации.

В C++ предусмотрено два типа комментариев. С двойной наклонной черты вправо (`//`) начинается комментарий, который называется комментарием в стиле C++. Такой комментарий указывает, что компилятор должен игнорировать все, что следует за наклонными чертами вправо до конца строки.

Наклонная черта, за которой следует звездочка (`/*`), открывает комментарий в стиле C. Встретив этот комментарий,

компилятор игнорирует все, что следует за этими символами, пока не найдет звездочку, за которой следует наклонная черта вправо (`* /`) — эти символы закрывают комментарий в стиле C. Комментарии в стиле C используются реже, за исключением случаев, когда комментарий занимает несколько строк.

Пробельные символы

Часто некоторые строки в программе нужно оставить пустыми. Поэтому символы перехода на новую строку часто называют пробельными. Пустые строки облегчают чтение программы, если отделяют различные разделы кода (связанные инструкции) один от другого. Строки 2 и 4 в нашей программе — пустые.

Иногда фигурная скобка используется почти для того же, что и пробельные символы (см. строки 6 и 10).

Функции

Хотя главная функция `main()` и является функцией, вызывается она не как обычно, а автоматически — тогда, когда вы запускаете программу. Вызовы всех других функций должны быть записаны в коде, и эти другие функции будут вызваны во время выполнения программы.

Программа выполняется строка за строкой, причем выполнение начинается с начала главной функции и продолжается до тех пор, пока не будет вызвана другая функция. В этом случае управление передается другой функции. Когда эта другая функция заканчивается, она возвращает управление на строку после ее вызова в главной функции. Если вызываемая функция в свою очередь вызывает еще одну функцию, то управление таким же образом возвращается той строке в вызывающей функции, которая следует за вызовом в вызывающей функции.

Даже если какая-либо функция в тексте программы определена раньше главной (`main`), она не выполняется до главной: сначала всегда выполняется главная функция (`main`).

Функция либо возвращает значение, либо не возвращает ничего — в этом случае часто говорят, что они возвращают пустое значение (`void`). Обратите внимание, что главная функция `main()` всегда возвращает целое число (значение типа `int`).

Инструкция cout: вывод слов

Именно инструкция cout фактически отображает результат программы на экране или в окне. Это — специальный объект из библиотеки iostream.

Символ операции << (два знака “меньше чем”) называется *оператором вставки*, который передает то, что следует за ним, объекту cout. Это одно из средств объектно-ориентированного программирования.

В данной программе после оператора вставки следует *литерал* (поскольку в кавычки заключаются строки-литералы). Так что в нашем случае за оператором вставки следует *строковый литерал* — строка символов, заключенная в кавычки. Сам литерал состоит из всего того, что находится между кавычками.

Второй оператор вставки вставляет конец строки (endl — end of line) после конца строкового литерала, поэтому если на дисплее будет отображаться еще что-нибудь, то оно начнется с новой строки. В различных операционных системах для представления конца строки используются различные символы или комбинации символов, но endl позволяет записать программу, которая работает на любой операционной системе. Библиотека iostream конкретной операционной системы “знает”, какими именно символами нужно заменить символ конца строки endl.

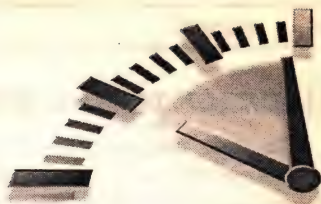
Вообще говоря, каждая строка, направляемая в cout, должна иметь endl.

Резюме

Вы научились отображать результаты выполнения программы (в нашем случае — литеральную строку), использовать важную стандартную библиотеку C++ (iostream), а также указывать пространство имен, в котором нужно разыскивать имена. Вы познакомились с новыми сообщениями компилятора об ошибках и научились использовать объект cout, оператор вставки << и манипулятор endl, чтобы отобразить строку символов.

УРОК 3

Вычисления



В этом уроке вы научитесь выполнять вычисления и отображать результаты вычислений на экране.

Выполнение вычислений и отображение их результатов

Теперь вы можете расширять пример программы из прошлого урока. На сей раз давайте просто изменим выводимую строку. Взгляните на листинг 3.1.

Листинг 3.1. Пример выполнения вычислений

```
1: #include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     // Должно напечатать число 6
8:     cout << ((6/2)+3) << endl;
9:     return 0;
10: }
```

ВЫВОД

Эта программа отображает значение 6, которое является результатом вычисления выражения $((6/2) + 3)$.

Выражения

$((6/2) + 3)$ — пример *выражения*. В этом случае выражение состоит из литералов (чисел), операторов (/ и +) и круглых скобок.

Порядок вычисления

Каждое выражение, в котором есть операторы, должно интерпретироваться компилятором. В самой простой интерпретации компилятор мог бы просто читать выражение слева направо. Если не учитывать круглые скобки, выражение выглядело бы так:

$6/2+3$

Если его вычислить, получится 6. Но если вычислить выражение

$3+6/2$

не учитывая старшинства операций, получится 4.5. Конечно, это неправильно. Ведь умножение и деление выполняются перед сложением и вычитанием, только в этом случае результат вычисления выражения

$3+6/2$

будет равен 6.

В языках C и C++ предусмотрено большое количество операторов (именно так часто называются знаки операции). Эти языки имеют обширный набор правил, которые определяют предшествование операторов. Операции с более высоким старшинством выполняются (в предшествующем выражении /), а их операнды (6 и 2) вычисляются раньше, чем операции (и операнды) с более низким старшинством (операторы вместе с операндами иногда называются *подвыражениями*).

Программисты часто не помнят всех этих правил. Если сомневаетесь, используйте круглые скобки. Иногда лучше использовать круглые скобки даже тогда, когда нет никаких сомнений.

Круглые скобки гарантируют определенный порядок вычислений. Подвыражения в круглых скобках вычисляются до вычисления остальных выражений. Например, в выражении

$3+(6/2)$

6 будет разделено на 2 прежде, чем 3 будет добавлено к результату деления.

Подробно старшинство операторов в C++ описано в приложении Б.

Вложенные круглые скобки

В сложных выражениях круглые скобки могут быть *вложенными*. (Это значит, что одно подвыражение находится внутри другого подвыражения.) Например:

$4 * (3 + (6 / 2))$

Фактически, при вычислении это сложное выражение приходится читать справа налево. Сначала нужно разделить 6 на 2, потом добавить 3 к полученному результату, а затем умножить результат сложения на 4.

Поскольку в C++ не требуется, чтобы выражение было написано в одной строке, можно ради удобства чтения использовать круглые скобки так, как часто используются фигурные скобки:

$3 + (6 / 2)$

При такой записи легче убедиться, что каждая открывающая круглая скобка имеет закрывающую круглую скобку. Это позволяет избежать обычных ошибок при записи выражений в программах.

Выражения в cout

`iostream` может отображать не только простой строковый литерал, но и результат вычисления сложного выражения. В примере программы выражение помещено в инструкцию `cout`, и число 6 выводится на экране так же просто, как строка "Hi there!" была выведена в уроке 2, "Вывод на пульт — стандартный вывод".

Все это работает потому, что компилятор строит программу таким образом, что выражение вычисляется перед выводом его результата с помощью инструкции `cout`.

Использование входного потока

Если вы используете какую-нибудь интегрированную среду разработки (IDE — Integrated Development Environment) вроде C++Builder фирмы Borland или Visual C++ от Microsoft, то при выполнении программ вроде нашей создается окно, в котором отображаются результаты (если они есть), причем это окно почти немедленно исчезает, когда программа завершает свое выполнение.

Если такое случается, в программе нужно сделать паузу до ее завершения, чтобы увидеть, правильные ли результаты выводит программа.

Даже если программы выполняются непосредственно из командной строки DOS или окна оболочки, ознакомьтесь со следующим кодом, поскольку в нем показан прием, позволяющий программе запрашивать информацию у пользователя.

```

1: #include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     cout << ((6/2)+3) << endl;
8:
9:     // Вы должны напечатать что-нибудь перед нажатием
    // клавиши Enter:
10:    char StopCharacter;
11:    cout << endl << "Press a key and \"Enter\": ";
12:    cin >> StopCharacter;
13:
14:    return 0;
15: }
```

Анализ

Строки 8–13 новые. Строки 8 и 13 — пустые, а строка 9 — это комментарий. Рассмотрим теперь строки 10–12 более подробно.

В строке 10 объявлена переменная. Эта переменная — место, в котором хранится один символ, который должен напечатать пользователь, чтобы закончить паузу. Тип переменной — слово `char` (символ) — указывает, что ожидается ввод символа. Так как в C++ каждая переменная должна иметь название (имя), то введенную переменную мы решили называть `StopCharacter`.

Строка 11 содержит строку (в смысле языка программирования). Эта строка будет отображена следующим образом:

Вывод

```

Press a key and "Enter":
Нажмите какую-нибудь клавишу и "Enter" ("Ввод"):
```

Обратите внимание, что наклонные черты влево (`\`) в строке 11 позволяют использовать кавычки внутри строкового литерала. Без наклонных черт `\` компилятор, увидев кавыч-

ку, предположил бы, что она обозначает конец строкового литерала. Попробуйте убрать их, и вы увидите, что произойдет. Вероятнее всего, вы получите сообщения об ошибках, когда попытаете откомпилировать программу.

В строке 12 ожидается, что пользователь введет один символ (это и есть способ организации паузы), Кроме того, эта же инструкция помещает введенный символ в `StopCharacter`. Для этой цели используется стандартный входной поток `cin`, определенный в `<iostream>`. Здесь используется оператор `>>`, который называется *экстрактором*, или *оператором извлечения*, поскольку он представляет собой как бы своеобразное “устройство” подачи или извлечения. Экстрактор указывает в направлении, противоположном вставке. Его направление указывает, что информация извлекается из `cin` и помещается в переменную.

Если теперь выполнить программу, она сначала напечатает 6. Затем она напечатает строку `Press a key and "Enter":` (Нажмите какую-нибудь клавишу и "Enter" ("Ввод")):. Такая вежливая просьба о вводе часто называется *подсказкой*. Затем программа будет ждать до тех пор, пока пользователь не нажмет на клавиатуре клавишу с символом, цифрой или знаком пунктуации, а затем и клавишу `Enter`. Когда это произойдет, программа возвратит 0 и остановится.



Как указано в формулировке подсказки и комментарии, одно только нажатие клавиши `Enter` не будет иметь никакого эффекта на программу. Вы должны сначала нажать другую клавишу.

Переменные

Ранее вы видели, что почти все в программе имеет название (имя). Конечно, литералы — исключение из этого правила. Они — то, чем они являются, и не имеют никакого названия (имени).

Переменные позволяют дать значению название (имя). Фактически это название места в памяти компьютера, в котором хранятся данные.

Определяя переменную в C++, вы должны сообщить компилятору не только ее название (имя), но и то, какая инфор-

мация будет храниться в определяемой переменной: число, символ (например символ `StopCharacter`) или что-то другое. Такая информация называется *типом* переменной. Помните, что C++ относится к языкам со строгим контролем типов, и идентификация типа переменной — необходимая часть строгого контроля типов.

Тип переменной сообщает компилятору, помимо всего прочего, какой объем памяти необходим для хранения значения переменной. Он также позволяет компилятору удостовериться, что переменная используется правильно (например, компилятор сгенерирует сообщения об ошибках, если вы попытаетесь делить число на символ).

Самая маленькая единица памяти, используемая для хранения переменной, называется байтом.

Размер памяти

В большинстве случаев размер символа равен одному байту, но в международных приложениях для хранения символа может потребоваться более одного байта.



Набор символов ASCII

Переменные типа `char` (символ) обычно содержат значения из набора символов ASCII. ASCII — набор из 256 символов, стандартизированный для использования на компьютерах. ASCII — акроним American Standard Code for Information Interchange (Американский стандартный код для информационного обмена). Почти все операционные системы, устанавливаемые на компьютерах, поддерживают ASCII. Однако на ASCII нельзя отобразить некоторые большие наборы символов, такие как набор японских иероглифов. В таких наборах для хранения символа требуется больше одного байта (поэтому символы таких наборов называются многобайтовыми) из-за большого количества букв в их алфавите. Для хранения таких символов часто используются переменные типа `wchar_t`.

Переменные типа `short int` (короткий `int`) занимают 2 байта на большинстве компьютеров, переменные типа `long int` (длинный `int`) — обычно 4 байта, а переменные типа `int` (без ключевого слова `short` (короткий) или `long` (длинный))

могут занимать 2 или 4 байта. Если программа рассчитана на Windows 95, Windows 98 или Windows NT, то `int`, вероятнее всего, будет иметь размер 4 байта, но иногда (очень редко, но тогда уж очень тщательно) приходится следить за этим.

Использование переменных и констант типа `int`

Переменная позволяет запомнить в программе результат вычисления, а не выводить его сразу же в инструкции `cout`.

```

1: #include <iostream>
2:
3: using namespace std; // пространство имен std
4:
5: int main(int argc, char* argv[])
6: {
*7:     const int Dividend = 6; // константа Делимое = 6
*8:     const int Divisor = 2; // константа Делитель = 2
*9:     // Результат = (Делимое / делитель)
*10:    int Result = (Dividend/Divisor);
*11:    Result = Result + 3; // к Результату добавили 3
*12:
*13:    cout << Result << endl;
14:    // Примечание:
15:    // Вы должны напечатать кое-что перед нажатием
    // клавиши Enter
16:    char StopCharacter;
17:    cout << endl << "Press a key and \"Enter\": ";
18:    cin >> StopCharacter;
19:
20:    return 0;
21: }
```

Анализ Строки 7–13 были изменены.

В строках 7 и 8 объявляются переменные с именами `Dividend` (Делимое) и `Divisor` (Делитель) и их значения устанавливаются равными 6 и 3, соответственно. Знак `=` называется оператором присваивания, операция присваивания помещает значение правой части в переменную, находящуюся в левой части. Данные переменные объявлены как имеющие тип `int`, который представляет число без десятичной точки.

Хотя делимое и делитель объявлены как переменные, поскольку они имеют названия (имена), слово `const` (константа)

в декларациях этих переменных поясняет компилятору, что программа не сможет изменить содержимое этих переменных никаким способом (в отличие от, например, переменной `Stop-Character`, в которой хранится любой символ, напечатанный пользователем). Переменные, в объявлениях которых применено ключевое слово `const` (константа), часто называют *константами*, или *постоянными* (надеюсь, вы помните, что π — название (имя) постоянной, чье значение примерно равно 3.14159).

В строке 10 объявлена переменная и ей присваивается результат вычисления правой части выражения. Для этого в выражении используются названия (имена) констант, объявленных в строках 7 и 8, так что значение результата `Result` зависит от значения этих констант.

Строка 11, возможно, самая трудная для непрограммистов. Помните, что переменная — поименованное место в памяти и что ее содержимое может изменяться. В строке 11 указано, что к текущему значению результата `Result` нужно добавить число 3 и поместить полученное значение в место, названное `Result` (Результат), при этом стирая то, что было там ранее.

В этом примере выводится 6. Это показывает, что можно изменять реализацию проекта (т.е. код программы), и все же получать тот же самый результат. Иными словами, чтобы решить задачу, программу можно писать по-разному.

Следовательно, иногда программу можно изменить, чтобы сделать ее более удобочитаемой или более удобной в сопровождении.

Типы переменных и допустимые названия (имена)

Целые числа бывают двух типов: со знаком и без знака. Ведь иногда нужны отрицательные числа, а иногда без них можно обойтись. Целые числа (`short` (короткие) и `long` (длинные)), если не указано, что они без знака, хранятся со знаком. Целые числа со знаком могут быть отрицательными или положительными (и нулем!), а целые числа без знака (`unsigned`) всегда положительны (точнее, неотрицательны).



Используйте тип `int` для числовых переменных

Во многих программах в большинстве случаев можно просто объявить простые переменные как целые (`int`) числа, поскольку они и в самом деле являются целыми числами со знаком.

Типы нецелочисленных переменных

Несколько типов переменных встроены в C++. Переменные таких типов удобно разделить на целочисленные переменные (уже знакомый тип), символьные переменные (обычно `char`) и переменные с плавающей запятой (`float` (одинарной точности) и `double` (двойной точности)).



Переменные с плавающей запятой

Переменные с плавающей запятой, в отличие от целых чисел, могут иметь дробные значения и десятичную точку.

Типы переменных, используемых в программах на C++, описаны в табл. 3.1. В этой таблице приведены типы переменных, указаны объем, обычно занимаемый ими в памяти, и диапазоны значений, которые могут храниться в этих переменных. Тип переменный определяет значения, которые могут быть сохранены в переменных данного типа, так что для примера взгляните на вывод из программы, приведенной в листинге 3.1.

Обратите внимание, что `e` в `3.4e38` (число в конце диапазона значений чисел с плавающей запятой) означает “умножить на десять в степени”, так что выражение должно читаться “3.4 умножить на десять в степени 38 (десять в 38-й степени)”, что равно 340 000 000 000 000 000 000 000 000 000 000 000.

Таблица 3.1. Типы переменных

Тип	Размер	Значения
<code>unsigned short int</code> (короткий <code>int</code> без знака)	2 байта	от 0 до 65 535
<code>short int</code> (короткий <code>int</code>)	2 байта	от -32 768 до 32 767
<code>unsigned long int</code> (длинный <code>int</code> без знака)	4 байта	от 0 до 4 294 967 295

Окончание табл. 3.1

Тип	Размер	Значения
long int (длинный int)	4 байта	от -2 147 483 648 до 2 147 483,647
char (символ)	1 байт	256 символьных значений
bool (логический, булев)	1 байт	true (истина) или false (ложь)
float (с плавающей точкой)	4 байта	от 1.2e-38 до 3.4e38
double (двойной точности)	8 байтов	от 2.2e-308 до 1.8e308

Строки

Строковые переменные — частный случай переменных. Они являются массивами и позже будут обсуждаться более подробно.

Чувствительность к регистру

C++ чувствителен к регистру. Это означает, что слова с различными комбинациями символов верхнего и нижнего регистра считаются различными. Переменная, названная age (возраст), — это не та же самая переменная, что Age (Возраст) или AGE (ВОЗРАСТ).



Набор не на том регистре — наиболее типичная ошибка, допускаемая программистами при записи программ на C++. Будьте внимательны.

Ключевые слова

Некоторые слова зарезервированы в C++, и их нельзя использовать как имена переменных. Это ключевые слова. Компилятор использует их при разборе программы. К ключевым словам относятся, например, if (если), while (пока), for (для) и main (главная). В справочной системе к компилятору должен быть приведен полный список ключевых слов, но, вообще говоря, любое разумное название (имя) перемен-

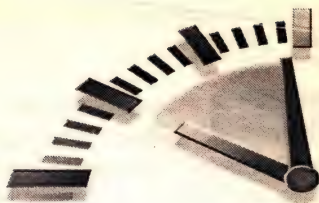
ной — почти наверняка не является ключевым словом. В этой книге приведен список наиболее часто используемых ключевых слов языка C++.

Резюме

Вы научились использовать литералы, а также объявлять и определить переменные и константы. Вы также научились присваивать значения переменным и константам, и узнали, какие значения и названия (имена) допустимы. Кроме того, вы вкратце узнали, как запросить информацию у пользователя, — именно это мы подробнее обсудим в следующем уроке.

УРОК 4

Ввод чисел



В этом уроке вы научитесь вводить данные для калькулятора и увидите, что происходит, если вводится не число тогда, когда ожидается ввод числа.

Ввод чисел

Программа, которая выполняет вычисления с предопределенными числами, полезна лишь для начала обучения, но большинство программ должно выполнять операции на данных, получаемых из внешнего мира. Так что давайте снова расширим пример. Обновленная программа показана в листинге 4.1.

Листинг 4.1. Ввод чисел пользователем

```
1: #include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     int Dividend = 1; // Делимое = 1;
8:     cout << "Dividend: "; // Делимое
9:     cin >> Dividend; // Делимое
10:
11:     int Divisor = 1; // Делитель = 1
12:     cout << "Divisor: "; // Делитель
13:     cin >> Divisor; // Делитель
14:     // Результат = (Делимое / делитель)
15:     int Result = (Dividend/Divisor);
16:     cout << Result << endl; // Результат
17:     // Обратите внимание:
18:     // Вы должны напечатать что-нибудь перед нажатием
19:     // клавиши Enter
20:     char StopCharacter; // Символ
21:     cout << endl << "Press a key and \"Enter\": ";
22:     cin >> StopCharacter;
23:
24:     return 0;
25: }
```

Анализ

Самая интересная часть находится в строках 7–13. Констант больше нет, вместо них числа вводятся в переменные, причем инструкции ввода похожи на инструкции в строках 20 и 21 (которые взяты из кода предыдущего урока). Программа теперь делает только деление, и потому она стала более простой и общей.

В строке 7 теперь создается переменная *Dividend* (Делимое). Делимое первоначально устанавливается равным 1 (или *инициализируется* значением 1). Даже если программа получает входные данные от пользователя, всегда рекомендуется установить начальные значения переменных, причем начальные значения переменных нужно выбрать некоторым разумным способом — это часто называется *программированием с защитой*.

В строке 8 для отображения подсказки пользователю используется поток стандартного вывода *cout*.

В строке 9 вводится число из стандартного входного потока и помещается в переменную *Dividend* (Делимое).

В строках 11–13 те же самые действия повторяются для переменной *Divisor* (Делитель).

А вот что получается при проверке:

ВВОД

Dividend: 6
Divisor: 3

Делимое: 6
Делитель: 3

ВЫВОД

2

Dividend: 5
Divisor: 3

ВВОД

Делимое: 5
Делитель: 3

ВЫВОД

1

В чем ошибка?

Хотя первый ответ правильный, но второй — нет. Если вы используете обычный калькулятор, вы увидите, что $5/3 = 1.6666667$. Так почему же программа выводит 1?

Ответ довольно прост, хотя он раздражает новичков. В программе используется тип `int`, что означает, что выполняется *целочисленное деление*. В результате целочисленного деления отсекаются все цифры справа от десятичной точки. Это обычно называется *усечением* (*truncation*).

Однако калькулятор можно сделать более точным, используя тип `float` (числа с плавающей точкой), в котором — в отличие от типа `int` — предусмотрено место для хранения цифр справа от десятичной точки.

Листинг 4.2. Ввод чисел с плавающей точкой пользователем

```
1: #include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     float Dividend = 1; // Делимое с плавающей
                          // точкой = 1;
8:     cout << "Dividend: "; // Делимое
9:     cin >> Dividend; // Делимое
10:
11:    float Divisor = 1; // Делитель с плавающей
                       // точкой = 1
12:    cout << "Divisor: "; // Делитель
13:    cin >> Divisor; // Делитель
14:    // Результат с плавающей точкой =
    // (Делимое / делитель)
15:    float Result = (Dividend/Divisor);
16:    cout << Result << endl; // Результат
17:    // Обратите внимание:
18:    // Вы должны напечатать что-нибудь перед
    // нажатием клавиши Enter
19:    char StopCharacter; // Символ
20:    cout << endl << "Press a key and \"Enter\": ";
21:    cin >> StopCharacter;
22:
23:    return 0;
24: }
```

Только строки 7, 11 и 15 отличаются от соответствующих в листинге 4.1, причем единственное изменение состоит в замене типа переменных — теперь тип переменной определяет ключевое слово `float`, а не `int`.

Теперь проведем испытания снова. Это называется регрессивным (возвратным) тестированием; оно помогает удостовериться, что в результате изменений (замен) программа работает правильно.

ВВОД

Dividend: 6

Divisor: 3

ВЫВОД

2

ВВОД

Dividend: 5

Divisor: 3

ВЫВОД

1.66667

Теперь калькулятор работает (по крайней мере, для конкретных данных, использованных при тестировании).

Но во всех испытаниях пока что использовались допустимые значения. Такие испытания называют испытаниями *в допустимых пределах*. Они важны, но не достаточны. Вы также должны провести испытания *с нарушением допустимых границ*.

Чтобы выполнить такое испытание, входные данные программы должны быть вне допустимого диапазона — в нашем случае вместо числа можно ввести, например, букву.

ВВОД

Dividend: a

ВЫВОД

Divisor:

1

Press a key and "Enter":

Как видим, при таких входных данных, наша программа не ждет, пока пользователь введет делитель. Она отображает неправильный ответ и не ждет от пользователя реакции на подсказку Press a key and "Enter":. Это, конечно, совсем не то, что нужно. Следующий шаг состоит в том, чтобы исследовать проблему и точно определить, что же происходит на самом деле.

Что же происходит не так, как надо?

Это — один из тех вопросов, которых программист боится больше всего. И все же именно с этим вопросом сталкивается каждый программист. Программа делает что-то неправиль-

но — да так неправильно, что ведет себя непредсказуемо. Как же узнать, что произошло?

В данном случае можно использовать инструкции `cout`, чтобы узнать, где программа “умерла”. С помощью этих инструкций нужно выводить сообщение о состоянии программы в начале каждого шага. Если сообщение о состоянии отсутствует, значит, программа остановилась как раз перед тем шагом, от которого не поступило сообщение о состоянии.

Чтобы эти отладочные инструкции показывали, что происходит на каждой стадии, программа должна делать паузу после того, как выводится очередное сообщение о состоянии, и требовать, чтобы пользователь ввел символ.

В листинге 4.3 показана программа примера с этими отладочными инструкциями.

Листинг 4.3. Пример с отладочными инструкциями

```
1: #include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     char DebugGoOn;
8:     cout << "Dividend..." << endl;
9:     cin >> DebugGoOn;
10:
11:     float Dividend = 1;
12:     cout << "Dividend: ";
13:     cin >> Dividend;
14:
15:     cout << "Divisor..." << endl;
16:     cin >> DebugGoOn;
17:
18:     float Divisor = 1;
19:     cout << "Divisor: ";
20:     cin >> Divisor;
21:
22:     cout << "Calculating..." << endl;
23:     cin >> DebugGoOn;
24:
25:     float Result = (Dividend/Divisor);
26:     cout << Result << endl;
27:
28:     cout << "Calculation done." << endl;
29:     cin >> DebugGoOn;
30: // Обратите внимание:
31: // Вы должны напечатать что-нибудь перед нажатием
    // клавиши Enter
```



```
32:     char StopCharacter;  
33:     cout << endl << "Press a key and \"Enter\": ";  
34:     cin >> StopCharacter;  
35:  
36:     return 0;  
37: }
```

Вывод И вот результат:

Dividend...

Dividend: a

Divisor...

Divisor... Calculating...

1

Calculation done.

Press some key and "Enter" to terminate the program:

Это подтверждает, что программа на самом деле “не умирает”, но ввод неподходящего символа, когда во входном потоке ожидается число, делает входной поток непригодным для выполнения остальной части программы.



Средства отладки

Отладка программы с помощью `cout` и `cin` чревата ошибками сама по себе, потому что добавление и удаление отладочных инструкций может повредить исходный код самым неожиданным образом, если программист сделает ошибку. Поэтому вместо этого лучше использовать отладчик (например тот, который входит в среду разработки).

Как же исправить ошибку? В следующем уроке мы обсудим этот вопрос.

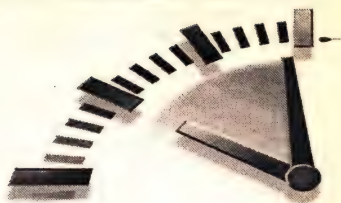
Резюме

Вы научились вводить числа в числовые переменные. Вы узнали о ловушках целочисленной арифметики и увидели, какие ужасные ошибки могут произойти, если программа получает непредусмотренную (или ошибочную) информацию от пользователя. В следующем уроке вы узнаете, как избавиться от подобных ошибок.

УРОК 5

Условные операторы

if и принятие решений в программах



В этом уроке вы научитесь использовать условный оператор if и булеву логику, обрабатывать ошибки и расширять программу так, чтобы она могла восстанавливаться после ошибок при вводе информации.

Обработка ошибок во входном потоке

Потребовалось два дня исследований и размышлений, чтобы придумать код, приведенный в листинге 5.1. Но, поскольку вы имеете только 10 минут, давайте сразу приступим к его изучению.

Листинг 5.1. Ввод чисел с обработкой ошибок во входном потоке

```
1: include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     int ReturnCode = 0;
8:
9:     float Dividend = 0;
10:    cout << "Dividend: ";
11:    cin >> Dividend;
12:
13:    if (!cin.fail()) // Делитель - число
```

```
*14:      {
*15:          float Divisor = 1;
*16:          cout << "Divisor: ";
*17:          cin >> Divisor;
*18:
*19:          float Result = (Dividend/Divisor);
*20:          cout << Result << endl;
*21:      }
*22:      else // Делитель - не число
*23:      {
*24:          cerr << "Input error, not a number?" << endl;
*25:
*26:          cin.clear(); // Сбросить биты ошибки
                        // входного потока
*27:      // Пропустить ошибочные данные,
      // чтобы продолжить работу
*28:          char BadInput[5]; // до 5 символов
*29:          cin >> BadInput;
*30:
*31:          ReturnCode = 1;
*32:      }
*33:      // Обратите внимание:
*34:      // Вы должны напечатать что-нибудь
      // перед нажатием клавиши Enter
*35:      char StopCharacter;
*36:      cout << endl << "Press a key and \"Enter\": ";
*37:      cin >> StopCharacter;
*38:
*39:      return ReturnCode;
*40: }
```

Ключ к этому коду находится в строке 13 — инструкция `if` (если).

Инструкция `if` (если)

Инструкция `if` (если) в строке 13 используется, чтобы принять решение, какая часть кода будет выполняться. Как обсуждалось ранее, программа начинает выполняться с начала главной функции и обычно заканчивается в ее конце, отклоняясь от этого порядка только при вызове функции, чтобы выполнить некоторое действие.

Однако программа, которая выполняет тот же самый код при каждом прогоне, не столь же гибка по сравнению с кодом, выполнение которого зависит от тех или иных обстоятельств.

Инструкция `if` (если) является ключевой в таком коде. Для принятия решения в этой инструкции используется логическое выражение, т.е. выражение типа `bool`.

Как принимать решение?

В стандарте ISO/ANSI, принятом в 1998 году, был введен специальный тип `bool` (названный в честь Джорджа Буля, знаменитого создателя булевой алгебры, которая используется при принятии решений в программах).

Этот новый тип имеет два возможных значения: `false` (ложь) и `true` (истина). Иногда они обозначаются как 0 и 1.

Каждое выражение может быть вычислено, чтобы определить его истинность (или ложность). Выражения, которые равны нулю, считаются ложными; все другие считаются имеющими значение `true` (истина).

Выражение в инструкции `if` (если) рассматривается как имеющее тип `bool`. Если выражение истинно, инструкция выполняет один набор инструкций, а если ложно — другой.

Инструкция `if` (если) имеет следующий формат:

```
/* логическим выражением называется выражение, */
/* значение которого приводится к типу bool */
if (/* логическое выражение */)
{
    // если истинно, выполняется эта часть кода
}
else
{
    // если ложно, выполняется эта часть кода
}
```

В строке 13 примера в инструкции `if` (если) используется логическое выражение `!cin.fail()`. (Логическое выражение в инструкции `if` (если) часто также называется *условием*.) Это выражение — результат применения операции `!` к вызову функции `fail()` объекта `cin`. Эта функция возвращает `true` (истина), если во входном потоке была ошибка.

Поскольку код, который выполняется, когда входные данные правильные, должен быть главным в исходной программе, в выражении-условии используется специальный оператор, который применяется к результату функции — оператор `!` (часто называемый *оператором отрицания* или *оператором не*). Оператор отрицания, получая истинный операнд, выдает значение `false` (ложь), а получая ложный операнд, выдает значение `true` (истина).

Оператор отрицания относится к префиксным одноместным (унарным) операторам. Такие операторы записываются

перед (поскольку они префиксные) своим единственным (поскольку они одноместные, или унарные) операндом. Этим они отличаются от оператора сложения (+), который является примером инфиксного оператора, т.е. оператора, который записывается между своими двумя операндами.

Оператор отрицания часто читается как “не”, поэтому выражение `!cin.fail()` можно читать как “не `cin.fail()`”.

Чтобы повысить удобочитаемость инструкции `if` (если), код в каждом из блоков имеет отступ. Кроме того, код в каждом из блоков заключен в фигурные скобки.

Восстановление после ошибки

Заметьте, что сообщение об ошибке посылается не в `cout`, но новому потоку, названному `cerr`. Это — стандартный поток для сообщений об ошибках, хотя почти всегда этот поток направляется на то же самое устройство, что и стандартный поток вывода `cout`, — поэтому сообщения об ошибках потока появляются на том же самом экране или в другом окне, если эти потоки отличаются. Как бы то ни было, сообщения об ошибках нужно посылать именно сюда.

Строки 24–28 позволяют восстановиться после ошибки: в строке 26 сбрасывается состояние потока, а затем “съедаются” символы, которые ждут в потоке ввода. Если этого не сделать, даже при состоянии потока, установленном как “не плохое”, следующие запросы к `cin` приведут к вводу ошибочных символов (символы, оставшиеся в потоке ввода, часто называются ожидаемыми символами) и программа перед окончанием выполнения не будет ожидать, когда пользователь должен будет сделать паузу.

Объявление `char BadInput [5]` в строке 28 создает место для хранения до пяти символов, которые следующее обращение к `cin` пытается получить из входного потока. Отказ в получении всех пяти символов еще не представляет условие (состояние) ошибки, так что эта часть кода может благополучно “съесть” целых пять ошибочных символов из входного потока, чтобы перед остановкой программы можно было сделать обычную паузу.

Кроме того, в главной функции `main()` значение переменной `ReturnCode` устанавливается равным 1 — на тот случай, если сценарий оболочки или пакетный файл захотят знать, что программа столкнулась с ошибкой. Программа больше не возвращает литерал, теперь она возвращает значение переменной `ReturnCode`.

В этом примере еще не рассматривался случай, когда делитель Divisor вводится неправильно. Код, представленный в листинге 5.2, делает и это.

Листинг 5.2. Ввод чисел и обработка ошибок во входном потоке для делимого Dividend и делителя Divisor

```

1: include <iostream>
2:
3: using namespace std;
4:
5: int main(int argc, char* argv[])
6: {
7:     int ReturnCode = 0;
8:
9:     float Dividend = 0;
10:    cout << "Dividend: ";
11:    cin >> Dividend;
12:
13:    if (!cin.fail()) // Делимое - число
14:    {
15:        float Divisor = 1;
16:        cout << "Divisor: ";
17:        cin >> Divisor;
18:
19:        if (!cin.fail()) // Делитель - не число
20:        {
21:            float Result = (Dividend/Divisor);
22:            cout << Result << endl;
23:        }
24:        else // Делитель - не число
25:        {
26:            cerr << "Input error, not a number?"
27:                << endl;
28:
29:            cin.clear(); // Очистить биты ошибки
30:                        // входного потока
31:
32:            // Проглотить ошибочные данные,
33:            // чтобы продолжить работу
34:            char BadInput[5]; // до 5 символов
35:            cin >> BadInput;
36:
37:            ReturnCode = 1;
38:        };
39:    }
40:    else // Делимое - не число
41:    {
42:        cerr << "Input error, not a number?" << endl;
43:
44:        cin.clear(); // Сбросить индикаторы ошибки
45:                    // при вводе
46:
47:        // Проглотить ошибочные данные,
48:        // чтобы приостановить программу

```



```

42:         char BadInput[5]; // до 5 символов
43:         cin >> BadInput;
44:
45:         ReturnCode = 1;
46:     };
47: // Обратите внимание:
48: // Вы должны напечатать что-нибудь перед нажатием
// клавиши Enter
49:     char StopCharacter;
50:     cout << endl << "Press a key and \"Enter\": ";
51:     cin >> StopCharacter;
52:
53:     return ReturnCode;
54: }

```

Проверка на ошибку при вводе делителя `Divisor` находится внутри блока, который выполняется, если условие во внешней инструкции `if` истинно. (Эта внешняя инструкция `if` (если) проверяла отсутствие ошибки при вводе делимого `Dividend`.) Эта “инструкция `if` (если) внутри другой инструкции `if` (если)” называется *вложенной инструкцией if* (если). Блоки для внутренней инструкции `if` (если) имеют отступ, чтобы сделать программу настолько удобочитаемой, насколько это возможно.



Количество уровней вложения инструкции `if` (если)

В программе очень легко сделать так много уровней вложения инструкции `if` (если), что программа будет нечитабельной, несмотря ни на какие отступы. Чтобы преодолеть эту проблему, используются функции.

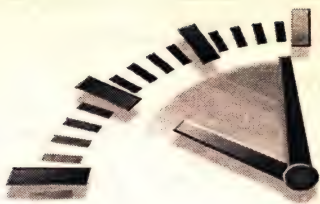
Обратите внимание также на то, что в C++ инструкция `if` (если) может иметь еще только ключевое слово `else`, но никаких других ключевых слов в ней не предусмотрено — этим C++ отличается от некоторых языков типа Pascal, в которых предусмотрено ключевое слово `then` (то). Несмотря на это блок, который выполняется тогда, когда условие в инструкции `if` (если) истинно, может называться *then-блоком*.

Резюме

Вы научились с помощью инструкции `if` (если) обнаруживать возникновение ошибок во входном потоке, а также сбрасывать ошибки в потоке и восстанавливать состояние после ошибок.

УРОК 6

Обработка исключений



В этом уроке вы узнаете, как использовать инструкции `try` и `catch` для обработки ошибок.

Обработка исключений — более лучший способ

Если взглянуть на полученный пример программы, не вникая в детали, можно заметить, что теперь доминирует код, обрабатывающий ошибки. Честно говоря, это не так уж необычно для профессиональных программ, однако переплетение кода, обрабатывающего ошибки, с обычным кодом создает определенные неудобства. Вы могли также заметить, что два `else`-блока содержат по существу тот же самый код, обрабатывающий ошибки.

В более ранних формах примера были предприняты некоторые шаги, чтобы отделить код, обрабатывающий ошибки, от кода, выполняющего обработку правильных данных. В частности, использовался оператор отрицания, чтобы поместить код обработки правильных данных поближе к началу программы, а код, обрабатывающий ошибки, — поближе к концу.

Однако в C++ есть лучший способ обработки ошибок — он называется обработкой исключений. При обработке исключения код, который сталкивается с ошибкой, вызывает исключение, которое *перехватывается* (часто говорят *ловится*) специальным кодом обработки исключения.

Исключения обрабатываются с помощью инструкций `try` и `catch`:

```
try
{
    // Код для обработки обычных, безошибочных данных
}
catch (/* декларация переменной-исключения */)
{
    // Код для обработки ошибок
}
```

В листинге 6.1 показано, как в нашем примере можно использовать обработку исключения.

Листинг 6.1. Применение обработки исключения для перехвата ошибочных данных

```
1: include <iostream>
2:
3: using namespace std; // станд. пространство имен
4:
5: int main(int argc, char* argv[])
6: {
7:     // Подготовка к вызову исключения в случае
    // ошибочных данных
*8:     cin.exceptions(cin.failbit);
9:
10:    int ReturnCode = 0;
11:
12:    try // обработка правильных данных
13:    {
14:        float Dividend = 0;
15:        cout << "Dividend: ";
16:        cin >> Dividend;
17:
18:        float Divisor = 1; // Делитель = 1
19:        cout << "Divisor: ";
20:        cin >> Divisor; // Делитель
21:        // Результат = (Делимое/Делитель)
22:        float Result = (Dividend/Divisor);
23:
24:        cout << Result << endl; // Результат
25:    }
26:    catch (...)//обработка исключений
27:    {
28:        cerr << // ошибка, не число
29:            "Input error - not a number?" <<
30:            endl;
31:
32:        cin.clear();// сброс состояния ошибки
33:
34:        // Проглотить ошибочные данные,
        // чтобы потом сделать паузу
```



```
35:         char BadInput[5]; // до 5 символов
36:         cin >> BadInput;
37:
38:         ReturnCode = 1;
39:     };
40:     // Обратите внимание:
41:     // Вы должны напечатать что-нибудь
42:     // перед нажатием клавиши Enter
43:     char StopCharacter;
44:     cout << endl << "Press a key and \"Enter\": ";
45:     cin >> StopCharacter;
46:     return ReturnCode;
47: }
```

При выполнении этого кода отображаются следующие результаты:

Вывод

```
Dividend: a
Input error - not a number?

Press a key and "Enter": .
```

Анализ

В строке 8 `cin` готовится к вызову исключения в случае ошибки. Эта инструкция специфична для библиотеки `iostream`, так как в ней используется специальная постоянная, которая указывает условие ошибки, т.е. состояние, которое вызовет исключение.

Строки 12 и 13 — начало кода, который может вызвать исключение. Если какая-либо строка `try`-блока или же любая вызываемая в нем функция, вызовет исключение, последующие строки этого блока будут пропущены, а исключение будет перехвачено в строке 26.

Где найти информацию, которая поможет перехватить исключение? Библиотеки — источник большинства исключений, и исключения, которые они вызывают, вообще говоря, документируются там же, где и классы библиотек или функции библиотек, которые их вызывают. Но исключения библиотеки `iostream` задокументированы не очень хорошо и расшифровка их названий (имен), как и определение их причин, требует некоторого времени для поиска в Internet.

Зачем использовать исключения?

Исключения, в отличие от инструкций `if`, могут показаться техническим приемом, но именно их следует предпочесть для обработки ошибок в C++. Хотя более старые библиотеки имели обыкновение сообщать об ошибках с помощью “кодов ошибок”, самые современные библиотеки вызывают исключения.

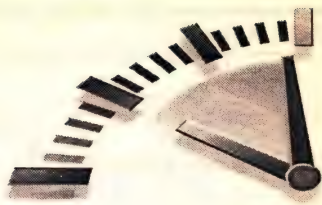
Кроме того, обработка исключений помогает очистить код. Хотя обработка ошибок — важный компонент любой реальной программы, программисты предпочитают сначала читать код обработки правильных данных, а потом — отдельно — код обработки ошибок, потому что так легче понять каждый из них. Использовать исключения — наилучший способ достичь этого.

Резюме

Вы научились использовать инструкции `try` и `catch` для обнаружения ошибок во входном потоке. Использование этих инструкций облегчает чтение кода. Оно также помогает избежать дублирования кода обработки ошибок.

УРОК 7

Функции



В этом уроке вы научитесь разбивать программу на несколько функций таким образом, чтобы каждая часть кода могла быть упрощена.

Что такое функция?

Когда люди говорят о C++, сначала они упоминают объекты. А объекты при выполнении работы полагаются на функции. Функция на самом деле представляет собой подпрограмму, которая может обрабатывать данные и возвращать результат. Каждая программа на C++ имеет, по крайней мере, одну функцию — главную, или `main()`. Когда программа запускается, ее главная функция `main()` вызывается автоматически. Главная функция `main()` может вызвать другие функции, некоторые из которых могут в свою очередь вызвать другие функции.

Каждая функция имеет свое собственное название (имя), и когда ее имя встречается в программе, управление выполнением программы переходит к телу этой функции. Этот процесс называется вызовом функции. Когда функция заканчивает выполнение ее кода, она возвращает управление, и выполнение продолжается со следующей строки вызывающей функции. Передача потока управления иллюстрируется на рис. 7.1.

Хорошо разработанная функция должна решать определенную задачу. Это означает, что она делает только одну вещь, а затем возвращает управление.

Сложные задачи должны быть разбиты на несколько более простых, а затем можно вызвать функции, предназначенные для решения каждой подзадачи. Такой код легче понять и сопровождать.

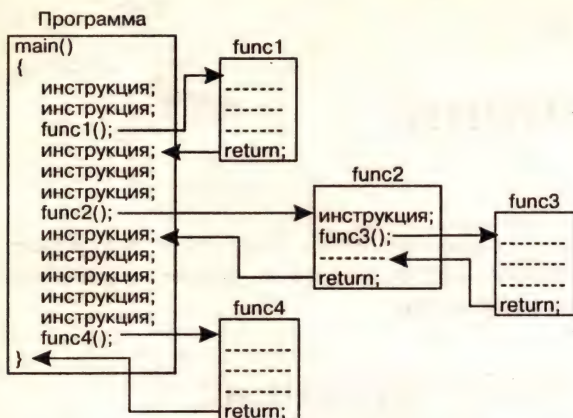


Рис. 7.1. Когда программа вызывает функцию, поток управления переключается на выполнение функции, а затем возвращается на строку после вызова функции

Определение функций

Прежде чем вызвать функцию, сначала нужно определить заголовок функции, а затем и тело функции.

Определение заголовка функции состоит из типа, возвращаемого функцией, ее названия (имени) и списка параметров. На рис. 7.2 показаны части заголовка функции.



Параметры или аргументы

Значения, передаваемые функции, называются ее аргументами. Аргументы бывают двух типов: формальные (в заголовке функции) и фактические (в вызове функции). Формальные аргументы также называют параметрами. Фактические аргументы — значения, передаваемые в момент вызова функции. Большинство программистов использует термины *параметр* и *аргумент* как взаимозаменяемые, но мы попробуем придерживаться указанных выше значений этих терминов.

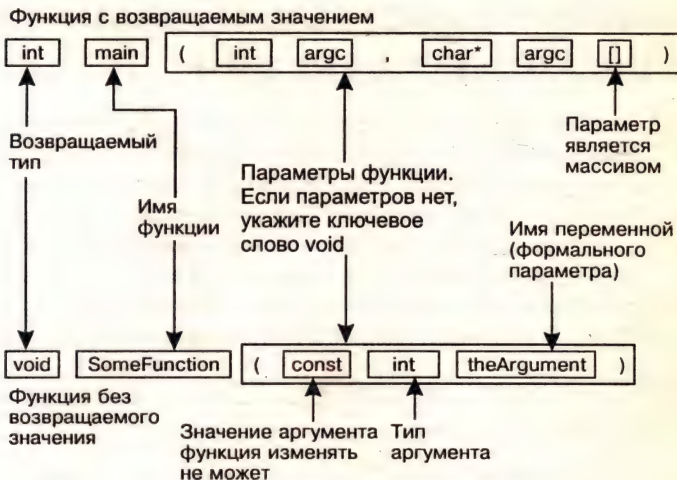


Рис. 7.2. Части заголовка функции

Тело функции — набор инструкций, заключенных в фигурные скобки. На рис. 7.3 показаны заголовок и тело функции.

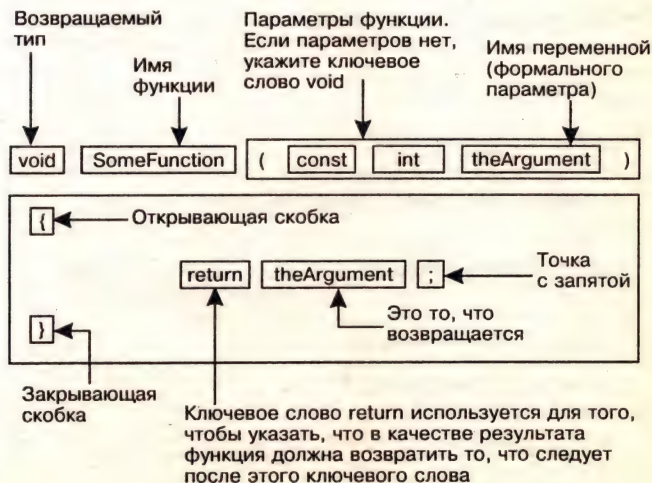


Рис. 7.3. Заголовок и тело функции

Разбиение программы примера на несколько функций

На данный момент наша программа стала довольно длинной. Даже несмотря на использование обработки исключения для уменьшения объема кода обработки ошибок, отслеживать ее становится все трудней. Так что лучше разбить ее на несколько функций, каждая из которых представляла бы собой значительно меньшую и более понятную часть кода, имеющую свое название (имя), которое указывало бы ее цель. В листинге 7.1 показано, как могла бы выглядеть программа.

Листинг 7.1. Пример с функциями

```
1: include <iostream>
2:
3: using namespace std; // станд. пространство имен
4:
5: void Initialize(void) // Нет возвращаемого значения
                       // и аргументов
6: {
7:     cin.exceptions(cin.failbit);
8: }
9:
10: float GetDividend(void) // Возвращает делимое
                        // с плавающей точкой
11: {
12:     float Dividend = 0;
13:
14:     cout << "Dividend: ";
15:     cin >> Dividend;
16:
17:     return Dividend; // Возвращает делимое
18: }
19:
20: float GetDivisor(void) // Возвращает делитель
                       // с плавающей точкой
21: {
22:     float Divisor = 1;
23:
24:     cout << "Divisor: ";
25:     cin >> Divisor;
26:
27:     return Divisor; // Возвращает делитель
28: }
29:
30: float Divide
```



```

30: (const float theDividend, const float theDivisor)
30: // Принимает неизменяемые параметры, возвращает
    // тип float
31: {
32:     return (theDividend/theDivisor);
    // Возвращает результат вычисления
32: // Возвращает результат вычисления
33: }
34:
35: int HandleNotANumberError(void) // Возвращает
    // код ошибки
36: {
37:     cerr <<
38:         "Input error - input may not have been
            a number." <<
39:         endl;
40:
41:     cin.clear(); // Сбрасывает состояние ошибки
    // потока
42:
43: // Проглотить ошибочные символы, чтобы потом сделать
    // паузу
44:     char BadInput[5];
45:     cin >> BadInput;
46:
47:     return 1; // Произошла ошибка
48: }
49:
50: void PauseForUserAcknowledgement(void)
51: { // Обратите внимание:
52: // Вы должны напечатать что-нибудь перед
    // нажатием клавиши Enter
53:     char StopCharacter;
54:     cout << endl << "Press a key and \"Enter\": ";
55:     cin >> StopCharacter;
56: }
57:
58: int main(int argc, char* argv[])
59: {
60:     Initialize(); // Вызывать функцию
61:
62:     int ReturnCode = 0;
63:
64:     try
65:     {
66:         float Dividend = GetDividend();
67:         float Divisor = GetDivisor(); // Делитель
68:
69:         cout << Divide(Dividend, Divisor) << endl;
70:     }
71:     catch (...)
72:     {

```

```

73:         ReturnCode = HandleNotANumberError();
74:     };
75:
76:     PauseForUserAcknowledgement();
77:     return ReturnCode;
78: }

```

Анализ

Сначала изучите главную функцию программы `main()` (строки 58–78). Почти весь код был удален из нее и заменен вызовами других функций, которые будут обсуждаться далее.

Функция без аргументов и возвращаемых значений

В строке 60 вызывается функция инициализации программы — в нашем случае она просто сообщает `cin`, какие условия (состояния) ошибки вызовут исключение. Она это делает точно так же, как в предыдущей версии.

```

void Initialize(void) // Нет возвращаемого значения
                      // и аргументов
{
    cin.exceptions(cin.failbit);
}

```

Это — простая функция, которая не принимает никаких аргументов и не возвращает никакого значения. Сравните ее с главной функцией `main()`, которая имеет аргументы и возвращает результат.

Помните, что `void` (пустой) означает “ничто, пустое пространство”. Так что функция ничего не возвращает и ничего не получает в качестве аргумента.

Функция без аргументов, но с локальными переменными и возвращаемым значением

В строке 66 выводится подсказка пользователю, чтобы он мог ввести делимое `Dividend`.

Вот функция, которая делает это:

```

float GetDividend(void) // Возвращает делимое с плавающей
                          // точкой
{
    float Dividend = 0;
}

```

```
cout << "Dividend: ";
cin >> Dividend;

return Dividend; // Возвращает делимое
}
```

Эта функция не принимает никаких аргументов, но возвращает результат. Отметим, что в ней объявлена локальная переменная `Dividend`, которая используется для того, чтобы инструкция `cin` могла куда-нибудь поместить введенное число. Эта переменная создается, когда функция получает управление, и исчезает, когда функция завершается. Инструкция возврата `return` помещает копию содержимого переменной во временное непоименованное место, из которого код главной функции `main()` сможет выбирать ее. Заметьте также, что названия (имена) локальных переменных в `GetDividend()` и в главной функции `main()` те же самые. Однако поскольку эти две функции независимы, такое допускается, и значения каждой из переменных могут быть разными.

Функция с аргументами и возвращаемым значением

В строке 69 вызывается функция `Divide` (Делить), причем ей передается делимое и делитель. Функция возвращает результат деления. Заметьте, что в ней не используются локальные переменные, хотя это и не запрещено правилами языка.

```
float Divide
{ (const float theDividend, const float theDivisor)
  // Принимает неизменяемые параметры, возвращает тип float
  {
    return (theDividend/theDivisor);
  }
  // Возвращает результат вычисления
}
```

Эта функция принимает два аргумента. Аргументы называются формальными, когда они находятся в заголовке функции. Они только резервируют место для любого фактического параметра — литерала, выражения или другого значения, которое передается функции, когда ее вызывают.

Чтобы развить эту идею, в программе следует придерживаться определенного соглашения об именах формальных ар-

гументов — они начинаются со строчных букв (theDividend, theDivisor). Благодаря этому аргументы будут легче отличить от локальных переменных при чтении кода. Поскольку аргументы представляют передаваемые функции данные, они — самый главный потенциальный источник проблем, и лучше всего сделать так, чтобы можно было быстро и легко увидеть, где они используются.



Соглашение

Несколько произвольное решение о последовательном способе сделать что-нибудь. Существуют, например, соглашения об именовании переменных и соглашения о структурах программ.

Вызов функции с аргументами, которые сами являются вызовами функций

Альтернативная форма для строки 69 может быть такой:

```
cout << Divide(GetDividend(), GetDivisor()) << endl;
```

Здесь вы снова можете увидеть, как действуют правила порядка вычислений, которые ранее уже обсуждались. В этом случае правила определяют, что сначала вызываются внутренние функции и что они возвращают свои результаты, которые затем передаются внешней функции в качестве аргументов. Порядок вызова функций определяется так же, как круглые скобки определяют порядок вычислений, так что количество уровней вызовов практически не ограничено.

Улучшение кода, или переразложение на классы

Процесс, который только что был выполнен — разделение кода на несколько отдельных функций — называют *рефакторингом*, *улучшением кода* и даже *переразложением на классы*. Это — критическая деятельность при создании сложной программы, особенно если проектирование начинается с более простой формы.

Многие программисты полагают, что усовершенствование (часто называемое расширением) или восстановление программы неизбежно делает программу более сложной и более тяжелой для понимания. Однако если при сопровождении используются возможности улучшения кода (рефакторинга, или переразложения на классы), это может действительно улучшить программу. И некоторые программисты (включая автора) полагают, что переразложение программ на классы заслуживает внимания само по себе, а совсем не обязательно в контексте расширения функциональных возможностей или очередного “ремонта”.

В нашем случае улучшение кода выполняется довольно просто. В примере последнего урока пробельные символы (пустые строки) в программе служили разделителями между функциями — фактически они указывали места, куда следовало бы вставить функцию.

Функции должны быть максимально *последовательными* (каждая инструкция ведет к цели, идентифицированной названием (именем) функции) и *минимально зависеть* от определенных фактических значений аргументов. При решении, должна ли строка кода быть перемещена в функцию, именно эти соображения играют важнейшую роль.



Унификация способа использования пробельных символов

Когда вы ищете раздел кода для переразложения, пробельные символы, независимо от их формы — будь то фигурные скобки или чистые строки, часто выделяют группу тесно связанных строк. Именно такие группы — наилучшие кандидаты для перемещения в отдельную функцию.

Где следует помещать код функций?

В C++ все необходимо объявить еще до использования. Таким образом, функция должна быть объявлена до ее вызова в другой функции. Из-за этого в программе на C++ функции, вызываемые главной функцией `main()`, обычно помещаются до нее. Функции, используемые функцией, вызванной из `main()`, определяются до вызывающей функции:

```
void A(void)
{
}
```

```
void C(void)
{
}
```

```
void D(void)
{
}
```

```
void B(void)
{
    C();
    D();
}
```

```
main()
{
    A();
    B();
}
```

Объявление — не всегда то же самое, что и определение. Чтобы объявить функцию, не определяя ее, в некоторых более старых программах на С и С++ используются прототипы функции.

Прототип — заголовок функции без тела. Он заканчивается точкой с запятой. В современных программах прототипы функций использовать приходится крайне редко.

Обычно начинать читать программу на С++ нужно с главной функции `main()`, и лишь затем выше главной функции `main()` искать вызываемые ею функции. Если такое возможно, те функции должны появляться в том порядке, в котором они вызываются. Например, в нашем примере порядок такой:

```
void Initialize(void)
float GetDividend(void)
float GetDivisor(void)
float Divide (float theDividend, float theDivisor)
int HandleNotANumberError(void)
void PauseForUserAcknowledgement(void)
int main(int argc, char* argv[])
```


По существу, это порядок, в котором они вызываются главной функцией `main()`.

Обратите внимание, однако, что формально в C++ этот порядок необязателен, и компилятор обычно “не жалуется”, если вы не в состоянии следовать этому соглашению.

Глобальные переменные

Можно иметь переменные, которые не находятся ни в какой конкретной функции. Такую переменную называют глобальной переменной, причем она может быть объявлена в начале модуля — файла с расширением `.cpp`.

Поскольку глобальная переменная не находится в какой-нибудь конкретной функции, она видима для всех функций. Это означает, что любая функция может получить (прочитать) или изменить (записать) ее значение.

Это увеличивает зависимость (сцепление) функций, использующих данную глобальную переменную, и, как следствие, затрудняет сопровождение программы. По этой причине рекомендуется избегать глобальных переменных, используя вместо них аргументы и возвращаемые значения.

Еще одна проблема с глобальными переменными возникает в случае, если локальные и глобальные переменные имеют то же самое название (имя). Тогда доступ к локальной переменной имеет более высокий приоритет, чем доступ к глобальной переменной. Поэтому любые изменения делаются в локальной, а не в глобальной переменной. В сложной программе, в которой используется много глобальных и локальных переменных, это может стать причиной “тонких” ошибок, затрудняющих отладку. Например, функция может случайно изменить содержимое локальной переменной тогда, когда программист хотел изменить глобальную переменную.

В проектах этой книги автор избегает использования глобальных переменных, но C++ не предотвращает их использование, и вы встретите их в программах, написанных и профессионалами, и любителями.

Испытание

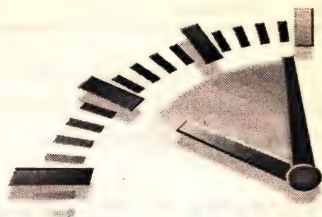
Не забудьте повторно провести все предыдущие испытания. Регрессивное (возвратное) тестирование является критическим — оно может свидетельствовать, что вы не повредили программу переразложением на классы.

Резюме

Вы узнали, как улучшить (переразложить на функции нескольких различных типов) готовую программу. Вам была продемонстрирована роль аргументов и локальных переменных. Вы имели возможность убедиться, что единство цели и зависимость — ключевые моменты в улучшении кода (переразложении на классы), и что пробельные символы могут помочь идентифицировать группы строк, которые будут выделены в отдельные функции, поскольку они (пробельные символы) помогают увидеть строки, которые находятся на подходящем уровне.

УРОК 8

Разделение кода на модули



В этом уроке вы узнаете, как разделить программу на файлы, которые могут компилироваться отдельно.

Что такое модуль?

Модули — файлы, которые могут быть откомпилированы отдельно. Результат компиляции этих модулей — набор выходных файлов компилятора. Эти файлы могут быть связаны компоновщиком в единую выполняемую программу.

В предыдущих уроках вы уже использовали модуль `iostream`.

Два файла — *заголовочный* (с расширением `.h`) и файл *реализации* (с расширением `.cpp`) — составляют в C++ модуль. `main.cpp` — единственное исключение из этого правила. Это — файл реализации, который никогда не имеет соответствующего заголовочного файла.



Реализация

Часть программы, которая содержит код, который может быть выполнен. В заголовке объявляются допустимые способы обращения к коду. Модуль `main.cpp` не имеет заголовка, потому что операционная система знает, что единственный способ обратиться к его коду состоит в том, чтобы вызвать функцию `main()`.

Вы уже использовали заголовочный файл `iostream.h`, `h`-файл, который вы включаете (с помощью `#include`) в начале главной программы `main.cpp`.

Зачем использовать модули?

Модули позволяют компилировать части программы отдельно. Это важно по следующим причинам.

1. Когда программа становится большой, перетранслировать нужно только те части, которые вы изменили; это экономит время.
2. Модули можно использовать совместно с другими программистами, причем другие программисты могут использовать созданный вами код таким же образом, как вы использовали библиотеку `iostream`.

Что находится в заголовочном файле?

Заголовочные файлы сообщают компилятору, что находится в модуле реализации, в них указываются названия (имена) функций и их параметры, а также определяются константы и типы данных.

Функции, перечисленные в заголовочном файле, *общедоступны*, потому что любой пользователь модуля может увидеть их. Файл реализации, однако, может содержать функции, не упомянутые в заголовочном файле, который не может использоваться чем-либо отличным от реализации модуля. Такие функции называются *частными* или *приватными*.

Создание заголовочного файла

Пришло время распределить несколько наших функций из примера по отдельным модулям. Давайте начнем с модуля `PromptModule.h`, который будет содержать функцию `PauseForUserAcknowledgement()` (листинг 8.1).

Листинг 8.1. Заголовок для `PromptModule`

```
1: #ifndef PromptModuleH
2: #define PromptModuleH
3:
```

```
4:     void PauseForUserAcknowledgement(void);  
5:  
6: #endif
```

Анализ

Строка 1 содержит специальную команду препроцессора. Она означает “if not defined” (“если не определено”). Когда название (имя), которое следует за `#ifndef` (это `PromptModuleH`), еще не определено в программе, строки, следующие за ней вплоть до строки `#endif`, будут переданы компилятору.

Без строки 2 строка 1 бесполезна. Именно в строке 2 определен символ `PromptModuleH`. Если бы заголовок был включен второй раз в той же самой программе, строка 1 заставила бы препроцессор проверить его список определенных символов, и он бы определил, что символ `PromptModuleH` был определен ранее. Поэтому код между строкой 1 и строкой 6 он не пропустил бы к компилятору второй раз.

(Обратите внимание, что строки 1, 2 и 6 никогда не попадут к компилятору. Они “съедаются” препроцессором.)

Почему строки 1, 2 и 6 настолько важны? Вообразите два библиотечных модуля, которые используют другой библиотечный модуль типа `iostream`. Вообразите, что оба этих модуля включены в главную программу `main.cpp`. Произошло бы дублирование заголовка, и компилятор увидел бы два объявления `cout` и не смог бы решить, какое из них выбрать. Поэтому обязательно используйте `#ifndef`, `#define` и `#endif` в каждом заголовке.

Давайте рассмотрим теперь строку 4 — ядро заголовочного файла. Это прототип функции, который уже встречался нам в объяснениях предыдущего урока. Вы знаете, что это прототип, потому что после него нет тела; он просто заканчивается точкой с запятой. Он содержит всю информацию, необходимую компилятору для того, чтобы проверить, правильно ли записан вызов функции и правильно ли выбраны типы параметров и ожидаемого возвращаемого значения.

Как выглядит файл реализации?

Файлы реализации во многом подобны главному файлу `main.cpp`, за исключением того, что они включают их собственный заголовочный файл так же, как любые другие, нужные им.

Стандартный C++ требует, чтобы файл реализации не был пуст. Но наличие чего-либо, кроме `#include`, для своего заголовочного файла необязательно, хотя именно файл реализации обычно содержит существенную часть кода.

Реализация для `PromptModule.h` представлена в листинге 8.2.

Листинг 8.2. Реализация для `PromptModule`

```
1: #include <iostream>
2:
*3: #include "PromptModule.h"
4:
5: using namespace std; // пространство имен std
6:
7: void PauseForUserAcknowledgement(void)
8: { // Обратите внимание:
9: // Вы должны напечатать что-нибудь перед нажатием
//клавиши Enter
10:     char StopCharacter; // Нажмите клавишу, а затем
//Ввод
11:     cout << endl << "Press a key and \"Enter\": ";
12:     cin >> StopCharacter;
13: }
```

Анализ

Строки 1 и 3 включают `iostream` и заголовочный файл модуля, строка 5 открывает пространство имен `std`, а сама функция идентична той, которая была в главном файле `main.cpp`.

Формат команды для включения `PromptModule.h` отличается от того, что обычно используется для `iostream`. Это обусловлено тем, что `iostream` — стандартный файл для включения и находится он в стандартном каталоге, в то время как `PromptModule.h` находится в текущем каталоге. Двойные кавычки указывают, что препроцессор должен искать заголовок в текущем каталоге, а если он его там не найдет, то нужно будет смотреть в любых других каталогах, к которым компилятор может получить доступ через свою командную строку (см. документацию к компилятору, чтобы научиться указывать дополнительные каталоги для поиска). При использовании этого формата команды включения в конце имени заголовочного файла должно быть расширение `.h`.

Как видите, если хотите использовать отдельную трансляцию, то в действительности слишком много изменений не потребуется. Но ради безопасности можно изменить и некоторые имена.

Изменение названий при создании библиотеки

Помните функцию `Initialize()`, которая использовалась для подготовки `cin` к вызову исключений? А что будет, если переместить обработку ошибок в ее собственный модуль, а некоторая другая библиотечная (или основная `main`) программа захочет использовать то же самое название (имя)? Это может создать проблему.

Вы уже знаете, как справиться с этой проблемой: нужно использовать пространство имен (инструкция `namespace`). Это решение используется в `iostream`, и оно вполне пригодно и для нашего случая.

Форма объявления пространства имен следующая:

```
namespace имя_пространства_имен
{
    Инструкции
}
```

Листинг 8.3 представляет собой новый заголовочный файл.

Листинг 8.3. Заголовок для `ErrorHandlingModule`

```
1: #ifndef ErrorHandlingModuleH
2: #define ErrorHandlingModuleH
3:
*4: namespace SAMSErrorHandling
*5: {
6:     void Initialize(void);
7:     int HandleNotANumberError(void); // Возвращает
                                         // код ошибки
*8: }
9:
10: #endif
```

Файл реализации приведен в листинге 8.4.

Листинг 8.4. Реализация для ErrorHandlerModule

```

1: #include <iostream>
*2: #include "ErrorHandlerModule.h"
3:
*4: namespace SAMSErrorHandling
*5: {
6:     using namespace std; // пространство имен std
7:
8:     void Initialize(void)
9:     {
10:         cin.exceptions(cin.failbit);
11:     }
12:
13:     int HandleNotANumberError(void) // Возвращает
                                     // код ошибки
14:     {
15:         cerr << "Input error, not a number?" << endl;
16:
17:         cin.clear(); // сбросить состояние ошибки
                     // в потоке
18:
19:         // Проглотить ош. данные, чтобы потом
         // приостановить программу
20:         char BadInput[5];
21:         cin >> BadInput;
22:
23:         return 1; // Произошла ошибка
24:     }
*25: }

```

Анализ

Объявленное в заголовочном файле (строки 4, 5 и 8 из листинга 8.3) пространство имен охватывает прототипы, а пространство имен, объявленное в файле реализации (строки 4, 5 и 25 из листинга 8.4), охватывает все функции.

Не забудьте прибавить объявление пространства имен к заголовку PromptModule. Пространство имен там будет называться SAMSPrompt.

Название (имя) пространства имен должно быть уникальным и одновременно осмысленным. Оно также должно создавать нечто осмысленное, когда присоединяется к названиям (именам) функций модуля, поскольку позже пространство имен будет использоваться как спецификатор имен функций в вызовах.

Вызов функций

Единственное различие в вызовах функций в новых модулях состоит в том, что необходимо квалифицировать имена функций их пространством имен. В листинге 8.5 показан новый главный файл `main.cpp`.

Листинг 8.5. `main.cpp` — вызов отдельно откомпилированных модулей

```
1: #include <iostream>
2:
*3: #include "PromptModule.h"
*4: #include "ErrorHandlingModule.h"
5:
6: using namespace std; // пространство имен std
7:
8: float GetDividend(void) // с плавающей точкой
9: {
10:     float Dividend = 0; // Делимое с плавающей
                          // точкой = 0
11:
12:     cout << "Dividend: ";
13:     cin >> Dividend;
14:
15:     return Dividend; // Делимое
16: }
17:
18: float GetDivisor(void) // с плавающей точкой
19: {
20:     float Divisor = 1; // Делитель с плавающей
                          // точкой = 1
21:
22:     cout << "Divisor: "; // Делитель
23:     cin >> Divisor; // Делитель
24:
25:     return Divisor; // Делитель
26: }
27:
28: float Divide // результат и аргументы
               // с плавающей точкой
29: (const float theDividend, const float theDivisor)
30: {
31:     return (theDividend/theDivisor);
32: }
33:
34: int main(int argc, char* argv[])
```



```

35: {
*36:     SAMSErrorHandling::Initialize(); // Инициализация
37:
38:     float ReturnCode = 0; // с плавающей точкой
39:
40:     try
41:     { // Делимое и делитель с плавающей точкой
42:         float Dividend = GetDividend();
43:         float Divisor = GetDivisor();
44:
45:         cout << Divide(Dividend, Divisor) << endl;
46:     }
47:     catch (...)
48:     {
*49:         ReturnCode =
50:             SAMSErrorHandling::HandleNotANumberError();
51:     };
52:
*53:     SAMSPrompt::PauseForUserAcknowledgement();
54:     return ReturnCode;
55: }

```

Анализ

В строках 36, 49 и 53 видно, что названиям (именам) функций предшествует название (имя) пространства имен, отделяемое двойным двоеточием от имен функций. Это двойное двоеточие называют *оператором разрешения видимости*, и этот оператор указывает, что название (имя) функции компилятор должен найти в указанном пространстве имен. Тот же самый оператор используется также для объектов и классов.

Конечно, указание пространства имен (и оператор разрешения видимости) нужно лишь в случае отсутствия инструкции использования пространства имен `using namespace`. Однако торопиться с ее добавлением не стоит, ведь инструкция использования пространства имен `using namespace` может свести на нет цель применения пространств имен, потому что она смешивает названия (имена) из всех пространств имен.

Заметьте также, что некоторые функции остаются в `main.cpp`; они не кажутся вероятными кандидатами на использование в других приложениях.

Раздельная компиляция

Нет никаких стандартов относительно названий (имен) команд вызова компилятора или компоновщика. Следующие команды компилируют модули и связывают их вместе, используя вымышленный компилятор (`cppcompiler`) и компоновщик (`cpplinker`) под Windows:

```
cppcompiler PromptModule.cpp
cppcompiler ErrorHandlingModule.cpp
cppcompiler Main.cpp
cpplinker Calculator.exe Main.obj PromptModule.obj
❖ ErrorHandlingModule.obj
```

Компилятор обычно генерирует *промежуточный* файл, чье название (имя) часто заканчивается расширением `.obj`, или `.o` (иногда называется *объектным* файлом даже несмотря на то, что это не имеет никакого отношения к объектно-ориентированному программированию). Компоновщик объединяет объектные файлы в *исполняемый* файл (под Windows файл с расширением `.exe`, под Unix имя файла не имеет расширения). Операционная система способна выполнить этот исполняемый файл.

Если теперь изменить только `ErrorHandlingModule`, его можно откомпилировать отдельно и связать с другими модулями:

```
cppcompiler ErrorHandlingModule.cpp
cpplinker Calculator.exe Main.obj PromptModule.obj
❖ (ErrorHandlingModule.obj)
```

Другие модули были откомпилированы ранее. В большой системе с десятками или сотнями модулей, это может сохранить время.

Команды компиляции и редактирования связей и их опции описаны в документации к компилятору.

Испытание

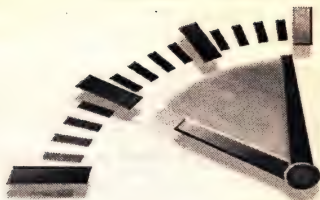
Разбиение программы на модули — форма улучшения программы, точнее, форма переразложения на классы. Не забудьте выполнять регрессивные испытания, чтобы удостовериться, что вы не повредили программу, разбивая ее на модули.

Резюме

Вы научились разбивать готовую программу на отдельно компилируемые модули и использовать `#include` в ваших собственных модулях, а также избегать проблем с использованием того же самого названия (имени) в нескольких модулях с помощью инструкции `namespace`. Кроме того, теперь вы умеете использовать оператор разрешения области для указания пространства имен, в котором находится нужная функция.

УРОК 9

Циклы: do и while



В этом уроке вы научитесь повторять выполнение части программы.

Что же у нас получилось и как ЭТИМ воспользоваться?

Вы изменили структуру программы для того, чтобы подготовить программу к увеличению размера и сложности. Теперь вы обобщите программу, чтобы с помощью цикла она могла выполнить любое число делений.

Повторение выполнения

В настоящее время каждый раз, когда вы хотите выполнить деление, вам приходится выполнять программу заново. К счастью, C++ предлагает несколько управляющих структур, подобных условному оператору, которые позволяют выполнять код в блоке неоднократно. Такие управляющие структуры называются *циклами*. В циклах используются логические выражения, которые управляют продолжением повторения. Благодаря им программа может выполнять свои инструкции много раз без остановки.



Управляющая структура

Любая инструкция, которая передает управление в программе. Обычный поток управления — сверху вниз в каждой вызванной функции. Условный оператор определяет, какому блоку передается управление: блоку `then` или блоку `else`. Инструкции `try` и `catch` служат для обработки особых ситуаций, когда вызвано исключение. Даже функция — управляющая структура, поскольку при ее вызове управление передается новой функции, а затем она возвращает его назад. В каждой управляющей структуре можно использовать по крайней мере один блок, заключенный в фигурные скобки.

Выполнение инструкций по крайней мере один раз

Во многих случаях инструкции нужно выполнить по крайней мере однажды, причем вполне возможно, что даже много раз. Именно для таких случаев предусмотрены циклы `do` и `while`.

Форма цикла `do`:

```
do
{
    инструкции
}
while (условие);
```

Давайте посмотрим, как этот цикл используется в главной функции `main()` нашего калькулятора, выполняющего все еще только деление (листинг 9.1).

Листинг 9.1. Цикл `do-while` в главной программе `main()`

```
1: int main(int argc, char* argv[])
2: {
3:     SAMSErrorHandling::Initialize();
4:
5:     do // По крайней мере один раз...
6:     {
7:         try
8:         {
```

```

9:         float Dividend = GetDividend();
10:        float Divisor = GetDivisor();
11:
12:        cout<< Divide(Dividend,Divisor) << endl;
13:    }
14:    catch (...)
15:    {
16:        SAMSErrorHandling::HandleNotANumberError();
17:    };
*18:    }
*19:    while (SAMSPrompt::UserWantsToContinue
*19:↵        ("More division? "));
20:
21:    return 0;
22: }

```

Анализ

Строки 5–6 и 18–19 представляют собой управляющую структуру — цикл.

Строка 5 указывает начало цикла, строка 6 — фигурная скобка, которая начинает блок, находящийся внутри цикла, строка 18 — фигурная скобка, которая заканчивает блок, находящийся внутри цикла, а строка 19 — выражение, которое определяет, продолжается ли повторение. Это выражение — вызов `SAMSPrompt::UserWantsToContinue()`, новой функции в `PromptModule`.

Вывод этой программы имеет следующий вид.

Вывод

```

Dividend: 2
Divisor: 3
0.666667

```

```

More division? - Press "n" and "Enter" to stop: y
Dividend: 6
Divisor: 2
3

```

```

More division? - Press "n" and "Enter" to stop: n

```

Как видите, после ввода данных и отображения результатов следует подсказка — вопрос о необходимости продолжать работу.

Условие цикла

`while` указывает, что цикл продолжается до тех пор, пока логическое выражение истинно. Этот цикл управляется результатом типа `bool`, возвращаемым функцией.

Функция `UserWantsToContinue`, добавленная к `PromptModule` в пространство имен `SAMSPrompt`, является относительно простой. Она получает сообщение как параметр и возвращает результат типа `bool`. Эта функция приведена в листинге 9.2.

Листинг 9.2. `UserWantsToContinue` в `PromptModule`

```
1: bool UserWantsToContinue
1: { (const char *theThingWeAreDoing)
2: {
3:     char DoneCharacter;
4:
5:     cout <<
6:         endl <<
7:         theThingWeAreDoing <<
8:         " - Press \"\n\" and \"Enter\" to stop: ";
9:
10:    cin >> DoneCharacter;
11:
12:    return (DoneCharacter != 'n'); // true если не "n"
13: }
```

Анализ

Большая часть кода этой новой функции вам уже знакома, она почти идентична функции `PauseForUserAcknowledgement()` в том же самом модуле и пространстве имен. (Вызов функции `PauseForUserAcknowledgement()` больше не делается в `main.cpp`, потому что нет никакой необходимости спрашивать дважды, хотите ли вы остановиться.)

Однако строка 12, куда функция возвращает результат вычисления логического выражения, отличается одной особенностью. Выражение имеет значение `true`, если введенный пользователем символ отличен от `n`. Обратите внимание, что оператор `!=` означает “не равно”.

Символьный литерал `'n'` заключен в одинарные кавычки — именно это указывает компилятору, что это символьный литерал, а не строка символов. Компилятор обнаружил бы ошибку, если бы в этом месте стояла строка. Это результат строго контроля типов в `C++`, и он гарантирует дополнительную безопасность.

Обратите внимание на формальный аргумент этой функции. Он описан как неизменяемый — `const char *`, что означает: “строка, которая не может быть изменена”.

Размещение инструкций `try` и `catch`

Блоки `try` и `catch` расположены в цикле. Это означает, что программа не будет останавливаться, даже если пользователь введет не число, потому что обработчик исключения отобразит свое сообщение и затем передаст управление на следующую строку, которая содержит условие продолжения цикла и находится в конце цикла.

Если вынести блоки `try` и `catch` вне цикла, получится код, приведенный в листинге 9.3.

Листинг 9.3. Блоки `try` и `catch` вне цикла

```
1: try
2: {
3:     do // По крайней мере однажды....
4:     {
5:         float Dividend = GetDividend();
6:         float Divisor = GetDivisor();
7:
8:         cout << Divide(Dividend, Divisor) << endl;
9:     }
10:    while (SAMSPrompt::UserWantsToContinue
10:↵        ( "More division? "));
11: }
12: catch (...)
13: {
14:     SAMSErrorHandling::HandleNotANumberError();
15: };
```

Если бы вы осуществили эту альтернативу и пользователь ввел букву вместо числа, управление было бы передано вне цикла, на строку 12, запустился бы код блока `catch`, управление далее передалось бы на строку после кода `catch` (строка 15) и программа продолжала бы останавливаться в конце главной функции `main()`. Не так должна работать наша программа, но в случае обработки *фатальной ошибки* программа не может благополучно продолжить выполнение и тогда может потребоваться именно такой код.

Выполняем ноль или большее количество раз

Тело цикла `do-while` выполняется по крайней мере один раз. Но что если нужно остановиться сразу же, не выполняя тела цикла ни разу?

Тогда следует использовать цикл `while`. Это потребует лишь незначительного изменения главной функции `main()` из листинга 9.1 — измененная функция показана в листинге 9.4.

Листинг 9.4. Цикл `while`

```
1: int main(int argc, char* argv[])
2: {
3:     SAMSErrorHandling::Initialize();
4:
5:     while (SAMSPrompt::UserWantsToContinue("Divide? "))
6:     {
7:         try
8:         {
9:             float Dividend = GetDividend();
10:            float Divisor = GetDivisor();
11:
12:            cout << Divide(Dividend, Divisor) << endl;
13:        }
14:        catch (...)
15:        {
16:            SAMSErrorHandling::HandleNotANumberError();
17:        };
18:    };
19:
20:    return 0;
21: }
```

Анализ

Здесь строки 5, 6 и 18 — самые интересные. (Фактически, программа сократилась на одну строку.)

Обратите внимание, что в этом цикле `while` помещается в начале. Теперь сразу же появляется подсказка для пользователя. Это требует изменить строку подсказки, потому что сразу после запуска программы не имеет смысла спрашивать пользователя “More division?” (“Хотите ли вы еще выполнять деление?”).

Если пользователь отвечает чем-нибудь, отличным от n, программа делает то, что указано в цикле. Если пользователь отвечает n, управление передается на строку 20 и программа останавливается.

Цикл с условием продолжения (while), в отличие от цикла повторения (do), заканчивается просто закрывающей фигурной скобкой (строка 18). В его конце нет никакого ключевого слова или условия, потому что и ключевое слово, и условие записаны в начале. Выполнение программы, содержащей цикл с условием продолжения (while), приводит к следующей выдаче:

ВЫВОД

```
Divide? — Press "n" and "Enter" to stop: y
Dividend: 2
Divisor: 3
0.666667
```

```
Divide? — Press "n" and "Enter" to stop: y
Dividend: 6
Divisor: 2
3
```

```
More division? Press "n" and "Enter" to stop: n
```

```
Делить? — Нажмите "n" и "Ввод", чтобы остановиться: y
Делимое: 2
Делитель: 3
0.666667
```

```
Делить? — Нажмите "n" и "Ввод", чтобы остановиться: y
Делимое: 6
Делитель: 2
3
```

Теперь программа с самого начала спрашивает пользователя, хочет ли он продолжать работу с программой.

В данном случае выбор между циклом do-while и циклом while — просто вопрос вкуса. Программу можно использовать и с циклом do-while, но в дальнейшем во многих случаях цикл с условием продолжения (while) будет предпочтительнее.

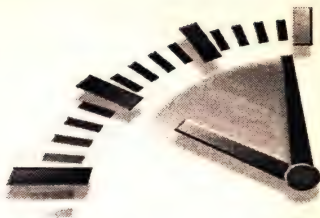
Резюме

Вы научились использовать циклы и исключения внутри них. Вы изучили циклы `do-while`, которые выполняют свое тело один или большее количество раз, а также циклы с условием продолжения (`while`), которые выполняют свое тело нуль или большее количество раз.

Урок 10

Вложенные циклы

и сложные логические выражения



В этом уроке вы изучите более сложные варианты принятия решений в вашей программе.

Вложение циклов

Так же, как условные операторы и другие управляющие структуры вроде `try` и `catch`, циклы могут быть вложены друг в друга. Но, как и в случае с условным оператором, это может быть ошибкой, и иногда лучше оптимизировать выполнение функций внутреннего цикла.

В качестве примера вы могли бы создать вариант функции `UserWantsToContinue()`. Она содержала бы цикл, который заканчивался бы только тогда, когда пользователь вводит у либо n. Никакой другой символ, отличный от у, не мог бы использоваться для обозначения “продолжать”. Такая функция показана в листинге 10.1.

Листинг 10.1. `UserWantsToContinueYOrN` в `PromptModule`

```
1: bool UserWantsToContinueYOrN
1:  (const char *theThingWeAreDoing)
2: {
3:     char DoneCharacter;
4:
5:     do
```



```

*6:    {
*7:        cout <<
*8:            endl <<
*9:            theThingWeAreDoing <<
10:            " - Press \"n\" and \"Enter\" to stop: ";
11:
12:        cin >> DoneCharacter;
13:
14:        if
15:            (
16:                !
17:                (
18:                    (DoneCharacter == 'y' )
19:                    ||
20:                    (DoneCharacter == 'n')
21:                )
22:            )
23:            {
24:                cout <<
24:                " . . .Error - " <<
24:                "please enter \"y\" or \"n\"." <<
24:                endl;
25:            };
26:    }
27: while
28:     (
29:         !
30:         (
31:             (DoneCharacter == 'y')
32:             ||
33:             (DoneCharacter == 'n')
34:         )
35:     );
36:
37: return (DoneCharacter != 'n');
38: }

```

Анализ

Здесь приведен общий образец. Сначала, в строках 7–12 — подсказка. Затем, в строках 14–22 — проверка правильности ввода; код в этих строках значит “введенный символ — не у или n” (оператор ! означает “не”, а оператор || — “или”). Если это действительно не так, строка 24 выводит сообщение об ошибке. Строки 27–35 проверяют условие снова, чтобы определить, нужно ли повторить подсказку. В этом разделе кода нужно использовать точно такое же условие, что и в инструкции if.

Поскольку в этих выражениях очень много вложенных круглых скобок, в коде они используются подобно фигурным скобкам. Хотя для этого требуется много строк, это предотвращает ошибки в сопоставлении круглых скобок и операторов: если идти прямо вниз от открытой круглой скобки, вы найдете соответствующую ей закрывающую круглую скобку. Легко, не так ли? Сравните это с альтернативой

```
((!(DoneCharacter == 'y') || (DoneCharacter == 'n')))
```

и вы увидите, почему дополнительные строки иногда делают программу понятнее.

В этом листинге нужно обратить внимание еще вот на что: инструкция `if` в строке 14 не имеет `else`. Если логическое выражение в этой инструкции ложно, управление передается на следующую инструкцию после блока, выполняющегося в случае истинности условия.

Операторы сравнения

В чем состоит различие между операторами “не” и “или”? Существует ли оператор “и”? Действительно ли полезно его применять? Не чрезмерно ли сложно все это? Нет, если ваш подход правильный и тщательно продуман.

Сначала давайте немного детальнее изучим простое логическое выражение. Помните, что логическое выражение может быть истинно или ложно. Есть шесть *операторов сравнения*, которые используются для сравнения значений. Подобно +, они являются инфиксными операторами, так что они имеют один операнд слева и один операнд справа. В табл. 10.1 приведены их названия, знаки, примеры их использования и примеры значений.

Таблица 10.1. Операторы сравнения

Название	Оператор	Пример	Значение
Равно	==	100 == 50;	false (ложь)
		50 == 50;	true (истина)
Не равно	!=	100 != 50;	true (истина)
		50 != 50;	false (ложь)

Окончание табл. 10.1

Название	Оператор	Пример	Значение
Больше чем	>	100 > 50;	true (истина)
		50 > 50;	false (ложь)
Больше или равно	>=	100 >= 50;	true (истина)
		50 >= 50;	true (истина)
Меньше чем	<	100 < 50;	false (ложь)
		50 < 50;	false (ложь)
Меньше или равно	<=	100 <= 50;	false (ложь)
		50 <= 50;	true (истина)

Есть также два инфиксных оператора, используемых в сложных логических выражениях — && (и) и || (или).

Результат операции && представляет истину, если ее оба операнда истинны; если же один или оба операнда ложны, результат представляет ложь:

(true && true) == true, (true && false) == false и (false && false) == false, т.е. (истина && истина) == истина, (истина && ложь) == ложь и (ложь && ложь) == ложь.

Результат операции || представляет истину, если истинен хотя бы один операнд:

(true || true) == true, (true || false) == true и (false || false) == false, т.е. (истина || истина) == истина, (истина || ложь) == истина и (ложь || ложь) == ложь.

И не забудьте о префиксном одноместном (унарном) логическом операторе ! (восклицательный знак). Оператор ! означает “не”. Если выражение истинно, ! делает его ложью; если выражение ложно, ! делает его истиной. Недаром этот оператор называется отрицанием!

Теперь подробнее рассмотрим сложные логические выражения в листинге 10.2. Как вы помните, нужно напечатать сообщение об ошибке и выполнить тело цикла, если пользователь вводит что-нибудь отличное от у или n. (Обратите внимание, что, когда используются круглые скобки, чтобы прочесть выражение, иногда приходится начинать с более внутренних подвыражений.)

Листинг 10.2. Условие подробно

```

*15:          (
*16:          !
*17:          (
*18:              (DoneCharacter == 'y' )
*19:              ||
*20:              (DoneCharacter == 'n')
*21:          )
*22:          )

```

Анализ

Строки 18 и 20 — самые внутренние выражения. Когда введен правильный символ, одно из выражений будет истинно. Когда введен ошибочный символ, ни одно из них не будет истинным.

Строка 19 — это “или”, которое объединяет эти два подвыражения в одно. Если хотя бы одно из выражений истинно (true) (пользователь вводит правильный символ), результат операции “или”, т.е. все выражение, будет истинно (true); если введен запрещенный символ, результат будет ложным (false).

Строка 16 отрицает результат, используя оператор ! (восклицательный знак). Причина этого может оказаться неожиданной, потому читайте внимательно.

Вспомните, что условный оператор и цикл исполняют свои блоки лишь тогда, когда условие истинно. Вы хотите, чтобы это условие было истинным (true), когда введен неправильный символ, но выражение истинно, когда входной символ правильный. Таким образом, необходимо отрицать истинный (true) результат, используя оператор ! (восклицательный знак). Это делает условие ложным при вводе правильного символа и истинным, когда символ ошибочный.

Если бы вы читали выражения так, как будто они записаны на естественном языке, понять все это было бы немного проще, а также проще было бы представить, как это работает:

```

If not (DoneCharacter == y or DoneCharacter == n)
then ошибка1.

```

Есть и альтернативная форма:

¹ Если не (DoneCharacter == y или DoneCharacter == n) тогда ошибка. — *Прим. ред.*

if (DoneCharacter != y **and** DoneCharacter != n) **then** ошибка².

В коде это выглядит так:

```
(DoneCharacter != 'y') && (DoneCharacter != 'n')
```

Это равносильно первой форме. Некоторые люди считают, что “и” в выражении такого стиля понимать немного сложнее. Выберите подходящий вам стиль и старайтесь последовательно придерживаться его.

Упрощение с помощью логических переменных

В предыдущем примере большая опасность заключается в том, что, поместив то же самое логическое выражение в два места, вы вызовете несчастный случай, если не будете следить за идентичностью этих двух выражений. Это может привести к серьезной трудноразрешимой проблеме во время эксплуатации программы. Чтобы избежать этого, можно использовать логическую переменную, как показано в листинге 10.3.

Листинг 10.3. UserWantsToContinueYOrN в PromptModule с логическим выражением

```
1: bool UserWantsToContinueYOrN
1:  (const char *theThingWeAreDoing)
2: {
3:     char DoneCharacter; // СИМВОЛ
*4:     bool InvalidCharacterWasEntered = false; // ЛОЖЬ
5:
6:     do
7:     {
8:         cout <<
9:             endl <<
10:            theThingWeAreDoing <<
11:            " - Press \"n\" and \"Enter\" to stop: ";
12:
13:            cin >> DoneCharacter;
14:
15:            InvalidCharacterWasEntered =
16:                !
17:                (
18:                    (DoneCharacter == 'y')
19:                )
```

² Если (DoneCharacter != y и DoneCharacter != n) тогда ошибка. — Прим. ред.

```

*20:                (DoneCharacter == 'n')
*21:                );
22:
*23:                if (InvalidCharacterWasEntered)
24:                {
25:                    cout <<
25:⚡                ". . .Error - " <<
25:⚡                "please enter \"y\" or \"n\"." <<
25:⚡                endl;
26:                };
27:            }
*28:            while (InvalidCharacterWasEntered);
29:
30:            return (DoneCharacter != 'n');
31: }

```

Анализ

Давайте начнем со строки 4, в которой определена логическая переменная ради сохранения результата вычисления логического выражения. Почему эта переменная, которая должна использоваться как условие цикла, определена вне цикла?

Важное правило C++ гласит, что переменная, объявленная в блоке (вспомните, блок — набор строк, заключенных в фигурные скобки), создается тогда, когда поток управления достигает открывающейся фигурной скобки, и уничтожается, когда управление передается вне блока. Если поместить эту логическую переменную в цикл `while`, ее нельзя будет использовать в условии цикла, так как в результате будет получено сообщение компилятора вроде следующего:

```

[C++ Error] PromptModule.cpp(34):
⚡E2451 Undefined symbol 'InvalidCharacterWasEntered'

```

Затем логическое выражение было удалено из условия в строки 15–21, в которых результат присваивается переменной.

В строке 23 инструкция `if` проверяет переменную, чтобы определить, была ли ошибка и нужно ли вследствие этого генерировать сообщение об ошибке.

В строке 28 проверяется та же самая переменная, чтобы решить, должен ли цикл повториться, выдавая пользователю подсказку снова.

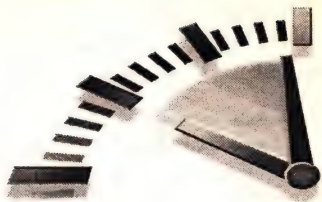
Конечно, это намного безопаснее, чем использовать два идентичных выражения. Кроме того, теперь выражение имеет название (имя переменной), так что его назначение намного легче понять.

Не забудьте добавить прототип к заголовку и изменить `main.cpp` таким образом, чтобы можно было использовать эту новую улучшенную функцию. И конечно, не забудьте провести регрессивные испытания.

Резюме

Вы научились использовать более сложные логические выражения и применять функции для упрощения вложенных циклов. Вы также узнали, как использовать логические переменные, чтобы избежать дублирования логических выражений в нескольких управляющих инструкциях (таких как инструкции `if` и циклы `do-while`) с целью сделать ваши программы более легкими для понимания и сопровождения.

УРОК 11



Переключатели (инструкции выбора `switch`), статические переменные и ошибки во время выполнения

В этом уроке вы изучите переключатели — инструкции выбора, или инструкции `switch`. Вы также научитесь использовать статические локальные переменные и вызывать исключение `runtime_error`.

Переключатели: инструкции выбора `switch`

Комбинации всевозможных “если” и “иначе” (инструкции `if` и `else...`) могут окончательно запутать вас, особенно если они вложены глубоко. В C++ им есть альтернатива. В отличие от `if`, в котором вычисляется одно значение, переключатель (инструкция выбора) позволяет выбрать ветвь вычислений в зависимости от множества различных значений, которые может (гипотетически) принимать некоторое выражение. Общая форма переключателя (инструкции выбора):

switch (выражение)

```
{
    case constantexpression1: инструкция; break;
    case constantexpression2: инструкция; break;
    ....
    case constantexpression3: инструкция; break;
    default: инструкция;
}
```

Здесь *выражение* — любое (правильное в C++) выражение, значение которого приводится к символу или другому простому результату (типа `int` (целое) или `float` (с плавающей точкой)), а *инструкции* — любые правильное в C++ инструкции или блоки.

Переключатель `switch` вычисляет *выражение* и сравнивает результат с каждым из значений *constantexpression*. Выражения *constantexpression* могут быть не только литералами (коими обычно они и являются), но и сколь угодно сложными выражениями вроде $3+x*y$, если только x и y — переменные-константы.

Если одно из значений *constantexpression* совпадает со значением *выражения*, управление передается на соответствующую *инструкцию* и вычисление продолжается до конца блока переключателя, пока не встретится оператор завершения `break`. Если одно из значений *constantexpression* не совпадает со значением *выражения*, управление по умолчанию передается на инструкцию `default`, если она имеется. Если инструкции перехода по умолчанию нет и ни одно из значений *constantexpression* не совпадает со значением *выражения*, вся инструкция выбора эквивалентна пустому оператору, не выполняющему никаких действий.



Заданный по умолчанию случай в переключателе (инструкции выбора)

Почти всегда в переключателе (инструкции выбора) желательно иметь метку `default` (заданный по умолчанию случай). Если какой-либо особой потребности в значении по умолчанию нет, его лучше всего использовать для проверки невозможности гипотетически невозможных случаев и генерации исключения, если все же такой случай произойдет.

Важно обратить внимание на то, что если в конце инструкции для какого-либо случая или блока нет `break`, управление будет передано на следующий случай. Отсутствие `break` иногда необходимо, но обычно оно является признаком ошибки, приводящей к удивительным эффектам. Если вы решили передать управление на следующий случай, обязательно поместите комментарий, указывающий, что вы опустили `break` умышленно, а не забыли его.

Обобщение калькулятора

Вы уже написали довольно много кода для выполнения деления. Теперь вы готовы сделать главный шаг и превратить эту простую программу в полноценный калькулятор. И переключатель (инструкция выбора `switch`) будет важной частью этого изменения.

Давайте начнем с главной функции `main()`, показанной в листинге 11.1.

Листинг 11.1. Главная функция `main()` реального калькулятора

```

1: int main(int argc, char* argv[])
2: {
3:     SAMSErrorHandling::Initialize();
4:
5:     do
6:     {
7:         try
8:         {
9:             char Operator = GetOperator() ;
10:            float Operand = GetOperand() ;
11:
12:            cout << Accumulate(Operator,
                                Operand) << endl;
13:        }
14:        catch (runtime_error RuntimeError)
15:        {
16:            SAMSErrorHandling::HandleRuntimeError
17:                (RuntimeError);
18:        }
19:        catch (...)
20:        {
21:            SAMSErrorHandling::HandleNotANumberError();
22:        }
23:    }
24: }
```

```

22:     }
23:     while (SAMSPrompt: :UserWantsToContinueYorN
    ↵                                     ( "More? "));
24:
25:     return 0;
26: }

```

Анализ

Изменения начинаются в строках 9 и 10, которые больше не хранят делители и делимые. Теперь программа может готовиться к использованию в качестве реального калькулятора, так что в тех строках хранятся “оператор” и “операнд”. Названия (имена) функций и переменных были тоже изменены соответствующим образом. Также обратите внимание, что `Operator` (Оператор) — отдельный символ `char`.

Строка 12 применяет оператор `Operator` к операнду `Operand` с помощью функции накапливающего сумматора `Accumulate()`. Эта функция “накапливает” текущее состояние вычисления и возвращает результат после обработки очередного нового оператора и операнда, чтобы отобразить результат в строке 12. Точно так же работают настольные и карманные калькуляторы.

Строки 14–17 перехватывают (`catch`) новый тип исключения — `runtime_error`, предусмотренный в Стандарте на библиотеку C++. Для этого объявляется переменная соответствующего типа внутри `catch()`. `catch` для этого исключения предшествует старому `catch`, потому что когда нужно перехватить определенный тип исключения, оператор его перехвата должен предшествовать `catch(...)`. Это необходимо потому, что троеточие (...) означает, что следующий блок обрабатывает все необработанные исключения.

Исключение `runtime_error` указывает, что пользователь ввел ошибочный символ оператора (например знак вопроса).

Вывод

```

Operator: +
Operand: 3
3

```

```

More? - Press "n" and "Enter" to stop: y
Operator: a
Operand: 3
Error - Invalid operator - must be

```

one of +, -, * or /

More? - Press "n" and "Enter" to stop: y

Operator: +

Operand: a

Input error - input may not have been a number.

More? - Press "n" and "Enter" to stop: y

Operator: -

Operand: 2

1

More? - Press "n" and "Enter" to stop: n

Переключатель (инструкция выбора switch)

В `GetOperator()` и `GetOperand()` нет ничего такого, чего вы не видели прежде в `GetDivisor()` и `GetDividend()` (за исключением того, что `GetOperator()` получает и возвращает символ), так что давайте пропустим эти функции и перейдем прямо к накопителю `Accumulate()` и переключателю (инструкции выбора `switch`), показанному в листинге 11.2.

Листинг 11.2. Накопитель `Accumulate()` в `main.cpp`¹

```
1: float Accumulate
1:  (const char theOperator, const float theOperand)
2: {
3:     static float myAccumulator = 0;
3:  // при запуске программы инициализируется 0
4:
5:     switch (theOperator)
6:     {
7:         case '+':
7:             myAccumulator = myAccumulator + theOperand;
7:             break;
8:         case '-':
8:             myAccumulator = myAccumulator - theOperand;
8:             break;
9:         case '*':
```

¹ Некоторые строки пронумерованы одним номером. Это значит, что автор рассматривает их как одну строку. Однако при редактировании обязательный в этих случаях знак переноса опущен. Это связано с тем, что обычно такие строки большинство программистов все же разбивает на несколько. — Прим. ред.


```
9:             myAccumulator = myAccumulator * theOperand;
9:             break;
10:         case '/':
10:             myAccumulator = myAccumulator / theOperand;
10:             break;
11:
12:         default:
13:             throw
14:                 runtime_error
15:                     ("Error - Invalid operator");
16:     };
17:
18:     return myAccumulator;
19: }
```

Анализ

Здесь в строках 5–16 появляется переключатель (инструкция выбора `switch`), которая ищет стандартный символ оператора и выполняет соответствующее действие на сумматоре для каждого операнда.

Любопытно, что при каждом вызове сумматор значение `myAccumulator` вычисляет не только исходя из фактических аргументов текущей функции, но еще и прибавляет вычисленное значение к сумме, накопленной в результате предыдущих вызовов `Accumulate()`.

Как он это делает?

Статические локальные переменные

В строке 3 объявлено, что сумматор (точнее, переменная `myAccumulator`) является статической переменной с плавающей точкой (**static float**). Обычные локальные переменные создаются при вызове функции и исчезают, когда управление возвращается вызывающей программе. В отличие от них, статические переменные создаются и инициализируются при запуске программы и не исчезают до останова программы. Фактически, эти переменные ведут себя точно так же, как глобальные, за исключением того, что они скрыты глубоко внутри функций, благодаря чему удается избежать ловушек, присущих использованию глобальных переменных.

Каждый раз, когда вы добавляете, вычитаете, умножаете или делите, берется то значение такой переменной, которое было вычислено в предыдущем вызове функции. Позже вы увидите, что эта идея очень подобна той концепции в объект-

но-ориентированном программировании, которая называется *инкапсуляцией* или *сокрытием информации*, и именно поэтому она обсуждается здесь.

Поскольку функция владеет этой переменной, но это не обычная локальная переменная, в имени переменной как напоминание используется специальный префикс “my” (“моя”). Для формальных параметров, как вы помните, используется префикс “the”. Это соглашение облегчает определение источника значения.

Самостоятельный вызов исключения

Строка 13 также имеет некоторую особенность. Она создает и вызывает исключение, если оператор не совпадает ни с одним из явно заданных значений в переключателе (инструкции switch). Используйте в этом случае инструкцию `throw` для вызова объекта-исключения `runtime_error` Стандартной библиотеки C++, причем передайте ему сообщение при обработке исключения. Взгляните на строку 14 главной программы `main()` — она перехватывает этот конкретный тип исключения.

Обработка нового исключения

Обработка этого нового исключения требует новой функции в пространстве имен `SAMSErrorHandling` модуля `ErrorHandlingModule`.

Эта функция, показанная в листинге 11.3, является довольно простой.

Листинг 11.3. `HandleRuntimeError()` в `ErrorHandlingModule`

```
1: int HandleRuntimeError(runtime_error theRuntimeError)
2: {
3:     cerr <<
4:         theRuntimeError.what() <<
5:         endl;
6:
7:     return 1;
8: };
```

Анализ

Строка 1 — заголовок функции, в нем показан формальный параметр `theRuntimeError`, который имеет тип `runtime_error`. Чтобы получить сообщение об ошибке, помещенное в объект-исключение в строках 14 и 15 функции `Accumulate()`, в строке 4 используется стандартная функция исключения `what()`. В строке 7 возвращается `int 1`, как и должно быть в функциях обработки ошибок. Возвращенное значение можно использовать для установки кода возврата в главной функции `main()`.

Естественно, прототип для этой функции должен быть добавлен к заголовку модуля — не забудьте сделать это!

Чтобы использовать `runtime_error`, необходимо изменить `ErrorHandlingModule`. Для этого следует включить в него обработку исключений. Для этого достаточно вставить команду `#include <exception>`. Кроме того, следует добавить инструкцию, указывающую на необходимость использовать пространство имен `std` везде, где используется исключение. Все это необходимо сделать и в заголовочном файле, и в файле реализации `ErrorHandlingModule`.

Резюме

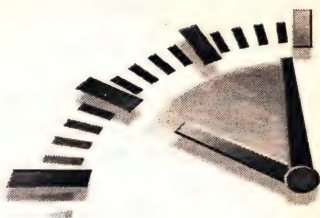
Вы теперь имеете полноценный калькулятор. Он позволяет вводить оператор и число, причем накапливает результаты вычислений в статической локальной переменной внутри некоторой функции. Чтобы определить, что делать с сумматором в зависимости от символа оператора, введенного пользователем, в калькуляторе используется переключатель — инструкция `switch`. Когда калькулятор получает недействительный оператор, он вызывает исключение Стандартной библиотеки C++ (`runtime_error`). Это исключение обрабатывается с помощью нового отдельного оператора перехвата `catch` для `runtime_error` в главной программе `main()`. Сообщение об исключении выводится с помощью новой функции в `ErrorHandlingModule`.

УРОК 12

Массивы,

циклы,

операторы приращения и декремента



В этом уроке вы научитесь создавать и использовать массивы, а также применять цикл `for` для обращения к элементам массива.

Использование массива для создания ленты калькулятора

В программе калькулятора используется массив, чтобы следить за тем, что было введено пользователем. Фактически, массив используется для создания “ленты калькулятора”, на которой для каждой введенной пары оператора и операнда предусмотрена отдельная запись.

Массив — поименованная последовательность мест для данных. Каждое место называется элементом и каждый элемент содержит один и тот же самый тип данных. Чтобы обратиться к элементу, нужно указать название (имя) массива и *индекс*, который является числовым смещением элемента в массиве; причем отсчет ведется от нуля.

Чтобы определить массив, необходимо указать тип данных, которые он будет содержать в каждом элементе, затем необходимо указать название (имя) массива и в качестве индекса (т.е. в квадратных скобках) количество элементов, ко-

торые он будет содержать. Вот пример объявления массива: `int SomeArray[3]`. Индексом (счетчиком элементов) может быть любой литерал, константа или постоянное выражение.

Чтобы получить или изменить значение его элемента, нужно указать название (имя) массива и в качестве индекса (т.е. в квадратных скобках) последовательный номер элемента (0 — первый элемент). Вот пример: `int x = SomeArray [0];`. Обращение к элементу может быть в левой или правой части инструкции присваивания — точно там же, где и переменная.

Лента Tape

Функция `Tape()` (Лента) довольно проста и закодирована в главном файле `main.cpp`.

Листинг 12.1. Функция `Tape()` (Лента) в главном файле `main.cpp`

```

1: void Tape(const char theOperator,
   const float theOperand)
2: {
3:     static const int myTapeSize = 3; // размер массива
4:
5:     static char myOperator[myTapeSize]; // часть для
                                           // Оператора
6:     static float myOperand[myTapeSize]; // часть для
                                           // Операнда
7:
8:     static int myNumberOfEntries = 0; // к-во записей
                                           // на ленте
9:
10: // В массивах отсчет элементов начинается с 0
11: // а индекс последнего элемента равен size - 1;
12:
13: if (theOperator != '?') // Добавить к ленте
14: {
15:     if (myNumberOfEntries < myTapeSize) // место
                                           // есть
16:     {
17:         myOperator[myNumberOfEntries] = theOperator;
18:         myOperand[myNumberOfEntries] = theOperand;
19:         myNumberOfEntries++;
20:     }
21:     else // массив может переполниться
22:     {
23:         throw runtime_error

```

```

23:                                     ("Error - Out of room on the tape.");
24:                                 };
25:                             }
26:                             else // Распечатать ленту
27:                             {
28:                                 for
29:                                 (
30:                                     int Index = 0;
31:                                     Index < myNumberOfEntries;
32:                                     Index++
33:                                 )
34:                                 {
35:                                     cout <<
36:                                     myOperator[Index] << ", " <<
37:                                     myOperand[Index] <<
38:                                     endl;
39:                                 }
40:                             };
41:     };
42: }

```

Анализ

Строки 3–9 настраивают ленту, т.е. устанавливают ее начальное состояние. Заметьте, что все локальные переменные являются статическими, подобно переменной `myAccumulator` в накопителе `Accumulate()`. Таким образом, они инициализируются при запуске программы и сохраняют свое значение от одного вызова до следующего.

В строке 3 указан максимальный размер массивов. При создании массивов требуется указать их размер с помощью постоянного выражения. В строке 3 указано значение этой постоянной и дано ей название (имя). Поскольку массивы являются статическими, постоянные, определяющие их размер, должны быть также статическими, так как в противном случае они бы не инициализировались при создании массивов в момент запуска программы.

В строках 5 и 6 объявлены массивы операторов и операндов, которые фактически и являются лентой. Вы могли бы сказать, что эти массивы *параллельны*, потому что оператором в первом элементе массива `myOperator` является тот, который применяется к операнду, находящемуся в элементе массива `myOperand` с тем же самым индексом.

В строке 8 объявлен счетчик количества элементов в массивах. Он нужен потому, что массивы имеют ограниченный размер, а значит, необходимо вызвать исключение, когда пользователь введет так много пар <оператор, операнд>, что произой-

дет выход за пределы массива (т.е. массив переполнится). В строке 32 этот счетчик обновляется.

Строки 10 и 11 вкратце напоминают правила языка C++.

В строке 13 проверяется, является ли `theOperator` тем символом, который запрашивает распечатку содержимого ленты. Если да, управление передается на строку 27.

В строке 15 проверяется, есть ли место на ленте. Если `myNumberOfEntries == myTapeSize`, индекс может выйти за верхнюю границу массива. Однако допустить этого нельзя, и потому в строке 23 вызывается исключение `runtime_error`, которое перехватывается в главной программе `main()`.

В строке 17 фактически начинается добавление к ленте новых записей. В этой строке присваивается значение соответствующему элементу каждого из массивов, причем индекс этих элементов на один больше индекса последнего заполненного элемента `myNumberOfEntries`. Этот трюк основан на том факте, что в C++ отсчет индексов массивов начинается с 0, а счет элементов ведется с 1, так что индекс нового элемента всегда равен счетчику уже занесенных элементов.

В строке 19 представлен новый, очень часто используемый арифметический оператор — двойной плюс (`++`). Это — *постфиксный оператор приращения (инкремента)*. Он добавляет 1 к предшествующей ему переменной. Эта строка обновляет счетчик `myNumberOfEntries`, чтобы в следующий раз можно было проверить, не будет ли выхода за границы массива.

Строка 28 находится в блоке, который выполняется, когда `theOperator` равен `?`, что означает “распечатать ленту”. В этом блоке для прохода по массиву используется цикл `for`; элементы массива отображаются с помощью объекта `cout`.

Цикл `for`

Цикл `for` объединяет три части — *инициализацию, условие и шаг* — в одну инструкцию в начале выполняемого этим циклом блока. Инструкция `for` имеет следующую форму:

```
for (инициализация; условие; шаг)
{
    инструкции;
};
```

Первая часть инструкции `for` — *инициализация*. Любые допустимые в C++ инструкции можно поместить в этой части (отделив помещаемые инструкции запятыми), но обычно инициализация используется для создания и инициализации индексной переменной. Наиболее часто такая переменная имеет тип `int`. Часть инициализации заканчивается точкой с запятой и выполняется только однажды перед запуском цикла.

Затем следует *условие*, которое может быть любым допустимым в C++ логическим выражением. Оно играет ту же самую роль, что и условие в цикле `while`. В нашем примере этот цикл выполняется до тех пор, пока индекс `Index` меньше чем `myNumberOfEntries`. Эта часть также заканчивается точкой с запятой, но выполняется она в начале каждого повторения.

В самом конце заголовка цикла расположен *шаг*. Как правило, именно здесь увеличивается или уменьшается индексная переменная, хотя здесь могут быть любые допустимые в C++ инструкции, разделяемые запятыми. Эта часть выполняется в конце каждого повторения.

Обратите внимание, что любая переменная цикла (чаще всего это индекс), объявленная в части инициализации инструкции `for`, может использоваться в цикле, но не вне его. В случае цикла в строке 28 в функции `Tape()`, переменная `Index` (Индекс) не может использоваться в строке 32, но может использоваться в строке 30.

Запись после конца массива

Когда вы получаете доступ к элементу массива, компилятор (к сожалению) не заботится, существует ли элемент фактически. Он просто вычисляет расстояние до первого элемента и генерирует инструкции, которые позволяют получить доступ к содержимому в указанном месте. Если элемент расположен вне границ массива, фактически он может оказаться чем угодно и результаты будут непредсказуемыми. Если повезет, программа потерпит крах немедленно или вызовет исключение в результате нарушения границ. Если не повезет, вы можете получить весьма странные результаты на существенно более поздней стадии выполнения программы. Вы должны удостовериться, что условие в инструкции `for` заставит цикл остановиться прежде, чем произойдет выход за границы массива.

Выражение-условие `Index < NumberOfElements` всегда гарантирует это, потому что самый большой допустимый индекс всегда на единицу меньше чем количество элементов в массиве.

Увеличение и уменьшение

Давайте рассмотрим немного подробнее строку 19 в функции `Tape()`.

Чаще всего к значению переменной приходится добавлять (или вычитать из него) 1, притом часто полученное значение снова присваивается этой же переменной. В C++ увеличение значения на 1 называется *инкрементированием*, *приращением* или просто *увеличением*, а уменьшение на 1 — *декрементированием* или просто *уменьшением*.

Оператор приращения (`++`) увеличивает значение своей переменной на 1, а оператор декремента (`--`) уменьшает его на 1. Таким образом, чтобы увеличить переменную `x` на 1, можно использовать такую инструкцию:

```
x++; // увеличиваем x
```

И оператор приращения, и оператор декремента имеют два стиля: префиксный и постфиксный.



Префиксные операторы

Префиксные операторы пишутся перед именем переменной (`++x`).



Постфиксные операторы

Постфиксные операторы пишутся после имени переменной (`x++`).

В простой инструкции не имеет значения, который из них вы используете, но в сложной инструкции, когда сначала переменная увеличивается (или уменьшается), а затем результат присваивается другой переменной, различие между префиксными и постфиксными операторами весьма существенно.

Префиксные операторы сначала изменяют значение (например, увеличивают его), а затем присваивают его. Постфиксные операторы позволяют сначала использовать старое

значение, а затем изменяют значение (например, увеличивают его) и присваивают новое значение источнику.

Другими словами, если переменная `x` типа `int` имеет значение 5, а в программе написано

```
int a = ++x;
```

то программа увеличит `x` (получится 6) и затем присвоит это значение переменной `a`. Таким образом, `a` будет иметь значение 6, и то же значение 6 будет иметь и `x`.

Если после выполнения этого записать

```
int b = x++;
```

то программа извлечет значение 6 из `x` и присвоит его `b`, а затем увеличит `x`. Таким образом, теперь `b` равно 6, а `x` теперь равно 7.

Это может показаться хитростью, но вы должны быть внимательны. Хотя компилятор не будет даже пытаться предотвратить смешивание префиксных и постфиксных операторов, человек, который сопровождает вашу программу, не всегда сможет достойно оценить ваше остроумие.

Лента калькулятора в сумматоре

Давайте рассмотрим теперь функцию `Accumulate()` в главном файле `main.cpp`. Как видно из листинга 12.2, в этой функции сделаны некоторые незначительные изменения, которые позволяют использовать ленту.

Листинг 12.2. Использование ленты в функции `Accumulate()`

```
1: float Accumulate
2: (const char theOperator, const float theOperand)
3: { // Инициализация при запуске
4:     static float myAccumulator = 0; // программы
5:
6:     switch (theOperator) // Переключатель
7:     {
8:         case '+': // Случай '+':
9:             myAccumulator = myAccumulator + theOperand;
10:            break;
11:
12:            case '-': // Случай '-':
```

```

13:         myAccumulator = myAccumulator - theOperand;
14:         break;
15:
16:     case '*':                // Случай '*':
17:         myAccumulator = myAccumulator * theOperand;
18:         break;
19:
20:     case '/':                // Случай '/':
21:         myAccumulator = myAccumulator / theOperand;
22:         break;
23:
24:     case '?':                // Случай '?':
25:         break;
26:
27:     default:                 // по умолчанию:
28:         throw
29:             runtime_error      // Ошибка
30:             ("Error - Invalid operator");
31: };
32:
33: Tape(theOperator, theOperand);    // Лента
34:
35: return myAccumulator;
36: }

```

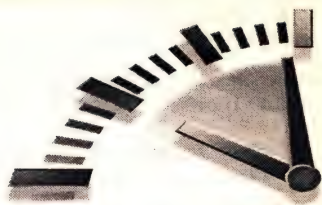
Анализ

В строке 33 на ленту записываются оператор и операнд. Строки 24 и 25 позволяют пользователю ввести оператор ?, который вызывает распечатку ленты.

Резюме

Вы убедились, что массивы могут быть очень полезны для создания переменных, которые составляют упорядоченную систему хранения данных, причем последовательность индексов определяет последовательность данных в массиве. Для создания ленты калькулятора установленного размера вы использовали два массива. Вы также изучили операторы приращения и декремента и цикл `for`. Все эти средства тесно связаны, потому что они совместно используются для обработки массивов.

УРОК 13



Память: динамическая память, стеки и указатели

В этом уроке вы научитесь распределять память и узнаете, в чем состоит различие между динамической памятью и стеком, а также освоите эффективное использование указателей.

Динамическая память и стек

Как вы помните, обсуждая локальные переменные и циклы, мы отметили, что открывающая фигурная скобка блока — то место в программе, где создаются переменные, определенные внутри блока, и что закрывающая фигурная скобка блока освобождает память, выделенную в этом блоке. Это же верно для функций, циклов и условных операторов.

Память для этих переменных выделяется в стеке.

Стек — обычная, подобная массиву структура данных, чьи элементы добавляются (заталкиваются с помощью операции `push`) в один конец, а затем вынимаются (вытаскиваются с помощью операции `pop`) с того же самого конца. Компилятор генерирует код, который обеспечивает рост (и сжатие) единственного стека программы каждый раз, когда происходит вызов функции (или возврат из нее), и каждый раз, когда открывается или закрывается блок.

На рис. 13.1 показано, что происходит, когда вы вызываете функцию `UserWantsToContinueYorN()`, которая имеет параметр, возвращаемое значение и две локальные переменные:


```

bool UserWantsToContinueYorN(const char
*theThingWeAreDoing)
{
    char DoneCharacter; // Символ
    bool InvalidCharacterWasEntered = false; // ложь
}

```

Перед вызовом функции резервируется место для значения, возвращаемого функцией, и для параметра функции.

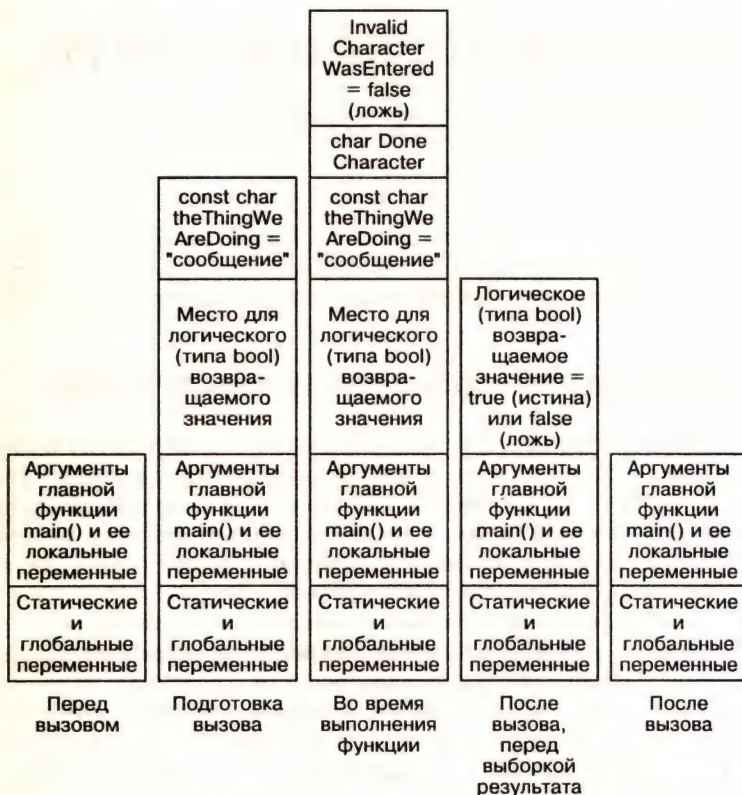


Рис. 13.1. Когда программа вызывает функцию, пространство для возвращаемого значения, фактических параметров и локальных переменных отыскивается путем проталкивания их в стек

Когда управление передается функции, обрабатываются определения локальных переменных и пространство для этих переменных отводится в стеке. Иными словами, когда управление передается функции, все ее локальные переменные заталкиваются в стек.

Наконец, наступает момент, когда выполнение функции завершается. Инструкция возвращения `return` помещает возвращаемое значение в зарезервированное для него место. Локальные переменные выталкиваются из стека. Управление возвращается главной функции `main()`. На место вызова функции подставляется возвращаемое значение, которое помещается в некоторую локальную или глобальную переменную или же отбрасывается. Место возвращаемого значения в стеке освобождается, так как само оно выталкивается вместе со всем остальным, что было записано при вызове.

Это типичное “распределение памяти в стеке”, причем очень аккуратное и удобное потому, что компилятор незаметно для программиста создает весь код, необходимый для очистки стека от ненужных переменных.

Для многих программ вполне достаточно распределения памяти в стеке. Однако в стеке нельзя поместить массив переменного размера. Единственный способ “изменять размеры” массива — создать новый массив желаемого размера, скопировать в новый массив содержимое старого массива, а затем освободить пространство, занятое старым массивом. После этого можно использовать новый массив вместо старого.

Память для этого не может выделяться в стеке потому, что распределение памяти в стеке для чего бы то ни было происходит только при входе в блок, а уничтожение чего бы то ни было — только при выходе из блока.

Возникшую проблему решает “динамическая память”, которая представлена на рис. 13.2.

Такая область памяти называется динамической памятью или кучей, потому что это — беспорядочная груда доступной памяти, причем в любое время могут использоваться только некоторые из ее ячеек.

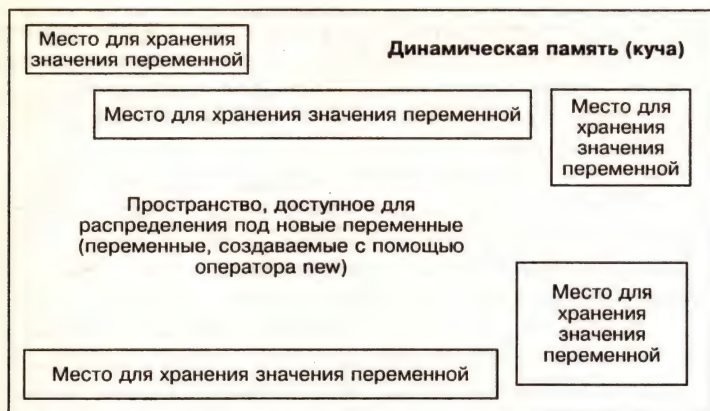


Рис. 13.2. Динамическая память — область, в которой выделяется пространство для новых переменных

Но как сообщить компилятору, что вы хотите получить память из динамически распределяемой области памяти, а не из стека? Для этого можно использовать специальный оператор `new` (новый, создать):

```
тип *имя = new тип;
```

Например:

```
// Символ *Something = новый символ;  
char *Something = new char;
```

Этот оператор резервирует место для одного символа в динамической памяти, причем он возвращает местоположение (адрес) зарезервированного пространства.

Указатели, ссылки и массивы

Проницательные читатели вспомнят, что нечто подобное предшествующим определениям они ранее видели в книге — например, в параметрах главной функции `main()`:

```
char* argv[]
```

и в параметре `UserWantsToContinueYorN()`:

```
char *theThingWeAreDoing
```


Что значит * в этих определениях? В определении * читается как “указатель на”, так что эти две строки можно читать как “указатель на массив символов” и “указатель на символ”.

На рис. 13.3 показано, как указатели “указывают” на место хранения в динамической памяти.



Рис. 13.3. Значениями указателей являются местоположения (адреса) в динамической памяти

Символ * является также префиксным оператором, который может использоваться вне определения, но только с указателями. Например:

```
// Символ *Something = новый символ
char *Something = new char;
*Something = 'x';
```

В этом случае символ * должен читаться как “память, на которую указывает”, так что мы получаем следующее: “память, на

которую указывает указатель `Something`, получает значение 'х', которое будет храниться в этой памяти". Символ `*` может также использоваться в правой части оператора присваивания, например:

```
char Other = *Something;
```

Это читается так: "объявить локальную переменную (т.е. переменную в стеке) `Other` и присвоить ей то же самое значение, что находится в памяти, на которую указывает `Something`". В этом смысле символ `*` называется оператором *разыменовывания*.

Хотя, как было показано в приведенных ранее примерах, в динамической памяти вполне возможно зарезервировать место для простых переменных, обычно делать этого не нужно. Указатели ведь на самом деле используются для представления массивов.

Массивы на самом деле представляют собой указатели

Система обозначений для массивов на самом деле является маскировкой для использования указателей. В C++, например, можно написать следующий код:

```
char SomeString[5] = "0123";
```

а затем получить доступ к `SomeString[0]` (иногда называемым "нулевым символом" или "нулевым элементом") с помощью инструкции вида

```
char FirstCharacter = *SomeString;
```

Эта инструкция присваивает содержимое нулевого символа из массива `SomeString` символьной переменной `FirstCharacter`; впрочем, подобных инструкций нужно избегать.

Интересно, что массив можно разместить в динамической памяти, на которую указывает указатель, и при этом использовать систему обозначений массива, чтобы обратиться к его элементам. Это означает, что можно создавать и уничтожать массив в динамической памяти. И в конечном итоге это означает, что можно изменять размер массива во время выполнения программы.

Массивы для ленты Tape () в динамической памяти

В листинге 13.1 показано, как можно распределять динамическую память для массивов в функции Tape () (Лента). Эта функция теперь переписана так, что использует динамическую память для хранения массивов myOperand и myOperator.



Длина символьной строки

Фактически длина символьной строки 0123 равна пяти символам: после 0123 подразумевается скрытый символ, который отмечает конец строки (строковый терминатор).

Строковый терминатор важен потому, что он используется многими функциями из библиотек C++ (в том числе и iostream) для определения конца строки.

Строковый терминатор имеет числовое значение 0. Но это не то же самое, что символ '0'.



Каждый символ имеет числовое значение

Набор символов ASCII определяет числовое значение каждого символа, причем эти числовые значения находятся в пределах от 0 до 255, символ '0' имеет числовое значение 48, а символ '\0' имеет числовое значение 0.

Символ \ в '\0' указывает, что следующее значение — специальное значение или числовое значение вроде 0, которое используется для создания соответствующего символа, или другое значение вроде одинарной или двойной кавычки, которую иным способом было бы невозможно поместить внутри кавычек, обрамляющих символьный или строковый литерал.



Как укоротить (обрезать) строку?

Чтобы укоротить (обрезать) строку, присвойте '\0' любому элементу массива, например: SomeString[3] = '\0';.

Листинг 13.1. Использование кучи в Tape()

```

1: void Tape(const char theOperator, const int theOperand)
2: {
*3:     static const int myTapeChunk = 3;
4:
*5:     static char *myOperator = new char[myTapeChunk];
*6:     static int *myOperand = new int[myTapeChunk];
7:
*8:     static int myTapeSize = myTapeChunk;
9:     static int myNumberOfEntries = 0;
10:
11: // массивы начинаются с элемента 0
12: // самый последний элемент имеет индекс размер - 1;
13:
*14:     switch (theOperator) // переключатель (выбор)
*15:     {
*16:         case '?': // случай '?': Распечатать ленту
17:
18:             for
19:             (
20:                 int Index = 0;
21:                 Index < myNumberOfEntries;
22:                 Index++
23:             )
24:             {
25:                 cout <<
26:                     myOperator[Index] << ", " <<
27:                     myOperand[Index] <<
28:                     endl;
29:             };
30:
31:             break;
32: // случай '.': программа завершается, удалить массив
*33:         case '.':
*34:
*35:             delete [] myOperator; // удалить
*36:             delete [] myOperand; // удалить
*37:
*38:             break;
39: // Добавить к ленте и расширить, если необходимо
*40:         default:
*41:
*42:             if (myNumberOfEntries == myTapeSize)
// расширить
*43:             {
*44: // Создать назначение для расширения
*45:
*46:             char *ExpandedOperator =
*47:                 new char[myNumberOfEntries +
myTapeChunk];
*48:

```

```

*49:             int *ExpandedOperand =
*50:             new int[myNumberOfEntries +
                               myTapeChunk];
*51:
*52: // Мы используем указатели, чтобы сделать
    // копию массива;
*53: // начинаем с начальной позиции.
*54:
*55:             char *FromOperator = myOperator;
*56:             int *FromOperand = myOperand;
*57:
*58:             char *ToOperator = ExpandedOperator;
*59:             int *ToOperand = ExpandedOperand;
*60:
*61: // Копировать старые массивы в новые
*62: // Это быстрее,
*63: // чем копирование массивов с помощью индексирования,
*64: // но это опасно
*65:
*66:             for
*67:             (
*68:                 int Index = 0;
*69:                 Index < myNumberOfEntries;
*70:                 Index++
*71:             )
*72:             {
*73:                 *ToOperator++ = *FromOperator++;
*74:                 *ToOperand++ = *FromOperand++;
*75:             };
*76:
*77: // Удалить старые массивы
*78:
*79:             delete [] myOperator; // удалить
*80:             delete [] myOperand; // удалить
*81:
*82: // Заменить старые указатели новыми
*83:
*84:             myOperator = ExpandedOperator;
*85:             myOperand = ExpandedOperand;
*86:
*87: // Записать новый размер массива
*88:
*89:             myTapeSize+= myTapeChunk;
*90:
*91: // Теперь можно добавить новую запись
*92:     };
*93:
*94:     myOperator[myNumberOfEntries] = theOperator;
*95:     myOperand[myNumberOfEntries] = theOperand;
*96:     myNumberOfEntries++;
*97: };
*98: }

```

Анализ

Вспомните, для чего предназначен этот код и что он должен делать. Большую часть времени он будет действовать точно так же, как старая лента `Tape()`. Единственный случай, когда будут возникать отличия, — это когда вы исчерпаете участок памяти на ленте. Когда это происходит, программа создает новые и большие массивы для `myOperator` и `myOperand`, копирует содержание старых массивов в новые массивы, избавляется от старых массивов и заменяет их указатели указателями на новые массивы. После этого программа продолжит выполнение, как будто ничего не случилось.

Изменения начинаются со строк 3, 5, 6 и 8, которые задают частоту замены массивов (`TapeChunk` — размер одного “куска” массива), а затем с помощью операции `new` создаются массивы в их начальном размере, т.е. с первоначально заданным числом элементов (это снова `TapeChunk`). В строке 8 запоминается текущий размер массива.

Новый код использует переключатель, а не управляющую инструкцию `if` (строка 14), потому что теперь есть три альтернативы: распечатка ленты (строка 16), удаление ленты в случае завершения программы (строка 33) или дозапись на ленту (строка 40) с возможным удлинением ленты, если в строке 43 будет обнаружено, что участок памяти исчерпан. Набор вложенных управляющих инструкций стал бы еще более неприемлемым для чтения.

Сконцентрируемся на логике расширения ленты (начинается в строке 43). Сначала создаются новые заменяющие массивы с помощью `new`, причем новое число элементов в этих массивах, расположенных в куче, указывается в квадратных скобках.

Строки 55–59, вероятно, выглядят немного странно. В этих строках объявляются указатели, которые будут использоваться для создания копий. Указатели устанавливаются так, чтобы они указывали на ту же самую память, на которую указывают старый и новый массивы, т.е. туда, где хранятся массивы. Эти дополнительные указатели позволяют избежать перерывов в адресации массива-источника и массива-адресата во время выполнения копирования (в строках 66–75).

Строки 66–75 — цикл `for`, предназначенный для копирования всех элементов старых массивов в новые.

Операцию копирования также можно записать следующим образом (листинг 13.2).

Листинг 13.2. Альтернативный способ копирования массивов-лент в Tape()

```
*66: for
*67:   (
*68:     int Index = 0;
*69:     Index < myNumberOfEntries;
*70:     Index++
*71:   )
*72:   {
*73:     ToOperator[Index] = FromOperator[Index];
*74:     ToOperand[Index] = FromOperand[Index];
*75:   };
```

Если бы использовалась эта альтернатива, строки 66–75 были бы не нужны. Так почему же не сделать это таким простым способом?

Выражение `ExpandedOperator[Index]` в действительности означает `(* (ExpandedOperator + (Index * sizeof (char))))`, т.е. “содержимое того, что находится на расстоянии (индекс `Index` умножить на размер символа) от начала массива”. Для больших массивов, такое большое количество умножений может существенно замедлить копирование. Поэтому многие программисты в C++ предпочитают копирование в стиле “указателя”.

Вот одна из инструкций копирования в этом стиле:

```
*ToOperator++ = *FromOperator++
```

Вспомните, что оператор `*` (также называемый *оператором разыменования*) имеет самое высокое старшинство и что постфиксный оператор приращения увеличивает значение не до, а после копирования. Тогда эта строка означает: “Скопировать содержимое элемента, на который указывает `FromOperator`, на место элемента, на которое указывает `ToOperator`. Затем переместить каждый указатель на следующий элемент”.

Строки 79 и 80 избавляются от старых массивов. Ключевое слово `delete` (удалить) — в противоположность `new` (новый, создать). Скобки указывают, что оператор `delete` (удалить) должен удалить массив, а не отдельный элемент массива.

Строки 84 и 85 переадресовывают указатели `myOperator` и `myOperand` на измененные местоположения массивов.

В строке 89 увеличивается размер — переменная, следящая за максимальным размером ленты; к нему добавляется число добавленных элементов. Для этого используется специальный оператор `+=`, который прибавляет значение правой части к значению левой части так, как будто вы написали `myTapeSize = myTapeSize + myTapeChunk`. (Такие же операторы присваивания со знаком равно `=` есть и для других арифметических операций: `-=`, `*` и `/=`.)

Строки 94 и 95 фактически прибавляют новые значения к массивам.

Ссылки

В C++ предусмотрена альтернатива указателям — *ссылки*. Ссылка походит на указатель, но не нужно указывать `*` для доступа к содержимому, на которое указывает указатель. В объявлении ссылки указывается `&` вместо `*`. Но ссылка — не указатель, а скорее дополнительное название (имя) переменной или места.

Вот как определяются и используются ссылки:

```
1: char &SomeReference = *(new char);  
2: SomeReference = 'x';  
3: delete &SomeReference;  
4: &SomeReference = *(new char); // Так нельзя!!!
```

Строка 1 размещает символ и заставляет ссылку обращаться к нему. Компилятор требует, чтобы в этой инициализации указатель был разыменован.

Строка 3 удаляет память, в которой хранится символ, т.е. фактически возвращает занятую память в динамическую память. Для этого используется префиксный оператор `&` (который читается как “местоположение” или “адрес”). Компилятор требует его наличия потому, что ссылка — не указатель на память, в которой хранится объект, а название (имя, псевдоним) для того участка памяти, в котором хранится объект.

Компилятор сгенерирует ошибку при разборе последней строки. Ссылка должна быть инициализирована там, где она определяется. После этого ее нельзя изменить, чтобы обра-

таться к чему-нибудь другому. Вот почему мы не используем в функции `Tare()` ссылки для изменения размеров нашей ленты-массива. Но ссылки превосходно подходят для формальных параметров.

Указатели опасны

Помните, что компилятору безразлично, пробуете вы получить значение или поместить его в элемент, который находится вне массива. Компилятор ведет себя так потому, что массивы — действительно удобная стенография для указателей.

Указатели столь же опасны, как и массивы, причем по тем же самым причинам. Взгляните на это:

```
char *SomePointer;  
*SomePointer = 'x'; // Кудааа...???
```

Как вы думаете, что делает этот код? Если вы отвечаете: “Я не знаю”, то вы правы. Поведение программы с неинициализированным указателем не определено и непредсказуемо:

```
char *SomePointer = NULL; // ПУСТОЙ УКАЗАТЕЛЬ  
*SomePointer = 'x'; // Кудааа...???
```

В этом случае `NULL` (ПУСТОЙ УКАЗАТЕЛЬ) не указывает ни на что. Если вы удачливы, программа остановится. Если нет, вся система может зависнуть. Однако `NULL` (ПУСТОЙ УКАЗАТЕЛЬ) часто используется, чтобы указать, что указатель не указывает на реальное место в динамической памяти. Это лучше, чем указывать на случайную ячейку в динамической памяти.

Удаление из динамической памяти

Операция удаления указателя, указывающего на место в куче, освобождает память, на которую он указывает. Операция удаления `delete` имеет две формы: с квадратными скобками `[]` (чтобы удалить массив) и без них (чтобы удалить обычную переменную). Удаление опасно. Вообразите, что случится, когда вы выполняете вот это:


```
char *SomePointer = NULL; // ПУСТОЙ УКАЗАТЕЛЬ
delete SomePointer; // удалить SomePointer
```

ИЛИ ЭТО:

```
char *SomePointer; // Символ
delete SomePointer; // удалить SomePointer
```

ИЛИ ЕЩЕ ВОТ ЭТО:

```
char *SomePointer = new char; // Символ *SomePointer =
                               // НОВЫЙ СИМВОЛ
delete SomePointer; // удалить SomePointer
delete SomePointer; // удалить SomePointer
```

ИЛИ ВОТ ТАКОЕ:

```
char *SomePointer = new char; // Символ *SomePointer =
                               // НОВЫЙ СИМВОЛ
delete SomePointer; // удалить SomePointer
char Stuff = *SomePointer; // СИМВОЛ = *SomePointer
```

Удаление массивов

Забыть удалить массив — очень распространенная ошибка:

```
char *SomePointer = new char[25]; // НОВЫЙ СИМВОЛ[25]
delete SomePointer; // удалить SomePointer
```

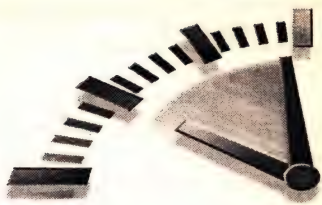
Этот фрагмент удалит нулевой элемент `SomePointer` и оставит другие 24 символа массива; они будут продолжать занимать место в динамической памяти. В конечном итоге свободного места в динамической памяти не будет, и программа прекратит выполнение. Это называется утечкой памяти.

Резюме

Это был один из наиболее трудных уроков в этой книге. Понимание указателей критично для овладения C++, но нельзя забывать, что они — причина большого количества проблем при программировании. Вы узнали, чем отличается распределение памяти в стеке от распределения динамической памяти, как получить память из динамической памяти, как ее использовать и как ее освободить. Вы видели некоторые из обычных ошибок и опасностей, и я надеюсь, что вы выработали серьезное отношение к мощи указателей и потенциальным проблемам, которые с ними связаны.

УРОК 14

Испытание, или тестирование



В этом уроке вы сделаете несколько изменений в калькуляторе и затем выполните некоторое испытание. Вы изучите стратегии тестирования программ на C++ и научитесь использовать их. Вы также познакомитесь с некоторыми особенностями языка C++.

Почему критичны испытания программ, использующих динамическую память?

Каждая программа должна быть проверена при каждом ее изменении. Но как вы видели в прошлом уроке, при использовании динамической памяти и указателей, опасность возникновения проблемы во время выполнения программы существенно возрастает. Только взгляните на некоторые из ошибок, с которыми можно столкнуться в такой программе:

- неинициализированный указатель;
- указатель, инициализированный значением NULL (ПУСТОЙ УКАЗАТЕЛЬ);
- повторное удаление (удаление дважды);
- удаление отсутствует вообще;
- выход за границы массива.

Эти проблемы могут вызвать тяжелые последствия где-нибудь около того места, где произошла ошибка, но таких последствий может и не быть. Иногда, например, при четвертом варианте вышеперечисленных ошибок, какой-либо очевидный эффект может вообще отсутствовать.

Компиляторы C++ побуждают к хорошему стилю программирования вроде строго контроля типов, но все же многие опасные ошибки остаются незамеченными. Существуют стратегии для создания программ, пригодных для тщательного тестирования, но никакое количество испытаний не позволит обнаружить все проблемы.

Обобщение калькулятора с помощью “небольшого языка”

Вы должны сделать небольшое изменение в интерфейсе пользователя калькулятора, чтобы сделать его более пригодным к тестированию.



Интерфейс пользователя

Часть программы, которая посылает выходную информацию пользователю и получает входную от пользователя.

Вместо подсказок, побуждающих пользователей ввести числа или указать, что они хотят остановиться, программа теперь будет получать команды на “небольшом языке”. Эти команды будут вводиться в программу во время выполнения, когда программа будет готова к вводу. Иными словами, все будет так, словно вы нажимаете клавиши на вашем карманном калькуляторе.

Инструкция на этом языке имеет следующую форму:

`<оператор><операнд>`

`<Оператор>` может быть одним из следующих:

- + прибавить операнд к сумматору;
- вычесть операнд из сумматора;
- * умножить сумматор на операнд;
- / разделить сумматор на операнд;


```

*25:         else if (Operator == '!' )
*26:         {
*27:             SelfTest(); // Самотестирование
*28:         }
*29:         else if (Operator == '.' )
*30:         {
*31:             // Не делать ничего, мы останавливаемся
*32:         }
*33:         else // Еще один оператор, нет операнда
*34:         {
*35:             Accumulator(Operator);
*36:         };
*37:     }
*38:     catch (runtime_error RuntimeError)
*39:     {
*40:         SAMSErrHandling::HandleRuntimeError
*40:         (RuntimeError);
*41:     }
*42:     catch (...)
*43:     {
*44:         SAMSErrHandling::HandleNotANumberError();
*45:     };
*46: }
*47: while (Operator != '.'); // Продолжать
*48:
*49: Tape('.'); // Сообщить ленте, что мы заканчиваем
*50:
*51: return 0;
*52: }

```

Анализ

В строке 5 переменная `Operator` (Оператор) объявлена вне цикла, потому что ее значение будет использоваться, чтобы остановить цикл, как видно из строки 47.

Строки 13–21 идентифицируют и получают операнды для операторов, имеющих их, а затем передают оператор и операнд сумматору.

Имя функции `Accumulate()` (Накапливать) было заменено на `Accumulator()` (Сумматор). Замена имени функции — существительное вместо глагола — указывает, что она имеет внутреннее состояние и что она зависит не только от параметров, передаваемых при ее вызове.

Строки 25–28 выполняют самопроверку.

Строка 29 гарантирует, что программа ничего не делает, когда введен оператор `..` Пустой блок проясняет это.

Строки 33–36 предназначены для обработки любого оператора без операнда. Этот вызов сумматора Accumulator() имеет лишь один параметр.

Изменения в сумматоре Accumulator()

В переключатель (инструкцию выбора switch) сумматора Accumulator(), код которого приведен в листинге 14.2, добавлены некоторые новые строки, потому что теперь на ленте с помощью Tape() регистрируется намного меньший процент операторов.

Листинг 14.2. Реализация новых операторов в сумматоре Accumulator()

```
*1: float Accumulator // Сумматор с плавающей точкой
1:  (const char theOperator, const float theOperand = 0)
2:  {
3:      static float myAccumulator = 0;
4:
5:      switch (theOperator)                // Переключатель
6:      {
7:          case '+':                        // Случай '+':
8:
9:              myAccumulator = myAccumulator + theOperand;
*10:         Tape(theOperator, theOperand); // Лента
11:         break;
12:
13:         case '-':                        // Случай '-':
14:
15:             myAccumulator = myAccumulator - theOperand;
*16:         Tape(theOperator, theOperand); // Лента
17:         break;
18:
19:         case '*':                        // Случай '*':
20:
21:             myAccumulator = myAccumulator * theOperand;
*22:         Tape(theOperator, theOperand); // Лента
23:         break;
24:
25:         case '/':                        // Случай '/':
26:
27:             myAccumulator = myAccumulator / theOperand;
*28:         Tape(theOperator, theOperand); // Лента
29:         break;
30:
31:         case '@':                        // Случай '@':
```



```

*32:
*33:         myAccumulator = theOperand;
*34:         Tape(theOperator, theOperand); // Лента
*35:         break;
36:
*37:         case '=': // Случай '=':
*38:             cout << endl << myAccumulator << endl;
*39:             break;
40:
41:         case '?': // Случай '?': распечатать ленту
42:             Tape(theOperator); // Лента
43:             break;
44:
45:         default:
46:             throw
47:                 runtime_error
48:                 ("Error - Invalid operator");
// Ошибка
49:     };
50:
*51:     return myAccumulator;
52: }

```

Анализ

В строке 1 сделано одно из наиболее существенных изменений. Теперь сумматор `Accumulator()` возвращает текущее значение `myAccumulator` в строке 51.

Новая особенность показана в формальном параметре `theOperand` в строке 1. Знак “=” и ноль следуют за именем формального параметра. Это означает, что формальный параметр является *необязательным*, причем если в вызове функции фактический параметр не указан, параметр получит значение по умолчанию — в нашем случае 0. Именно благодаря значению по умолчанию компилятор допускает вызов в строке 35 главной функции `main()`.

Другие новые строки ничего хитрого не содержат — они просто добавляют вызовы функции `Tape()` для записи на ленту каждого оператора, чье действие должно быть зарегистрировано, или реализуют новые операторы, вроде установки значения на сумматоре в строках 31–35 и отображения значения сумматора в строках 37–39.

Изменения в функциях ввода

Получение оператора и операнда требует незначительного изменения — как видно из листинга 14.3, подсказка была удалена.

Листинг 14.3. GetOperator() и GetOperand() без подсказки

```

1: char GetOperator(void)
2: {
3:     char Operator; // Оператор
4:     cin >> Operator; // Оператор
5:
6:     return Operator; // Оператор
7: }
8:
9: float GetOperand(void)
10: {
11:     float Operand; // Операнд
12:     cin >> Operand; // Операнд
13:
14:     return Operand; // Операнд
15: }
```

Функция самотестирования SelfTest

Функция самотестирования SelfTest (листинг 14.4) вызывается в строке 27 главной функции main(), когда вы вводите !. В этом случае запускается испытание сумматора.

Листинг 14.4. Функция самотестирования SelfTest()

```

1: void SelfTest(void) // Самотестирование
2: {
3:     float OldValue = Accumulator('='); // Сумматор
4:
5:     try
6:     {
7:         if
8:         (
9:             TestOK('@',0,0) &&
10:            TestOK('+',3,3) &&
11:            TestOK('-',2,1) &&
12:            TestOK('*',4,4) &&
13:            TestOK('/',2,2)
14:        )
15:        { // Испытание закончено успешно
16:            cout << "Test completed
17:                successfully." << endl;
```

```

18:         else
19:         { // Неудача на испытании
20:             cout << "Test failed." << endl;
21:         };
22:     }
23:     catch (...)
24:     {
25:         cout << // во время испытания произошло
                // исключение
25:↵         "An exception occurred during self test." <<
25:↵         endl;
26:     };
27:
28:     Accumulator('@', OldValue); // Сумматор
29: }

```

Анализ

Эта функция обернута в `try` и `catch`, так что если испытание сталкивается с проблемой, которая вызывает исключение, функция может восстановить состояние всех вещей таким, каким оно было перед началом испытания. Она может также поймать ошибки, происходящие из-за использования кучи в `Test()`. Но помните, что ошибки распределения кучи могут сделать такое большое повреждение, что исключение никогда не сможет быть вызвано.

Строка 3 сохраняет значение сумматора, которое восстанавливается в строке 28.

Строки 7–14 фактически выполняют испытания. Этот блок проверяет каждый оператор, который изменяет сумматор, вызывая функцию `TestOK()`. Поскольку здесь используется логический оператор `&&`, если какие-либо испытания терпят неудачу, то неудачей заканчивается вся самопроверка.

Функция `TestOK()`

Чтобы определить, получился ли на сумматоре `Accumulator()` ожидаемый результат для каждой поданной на него пары <оператор, операнд>, `SelfTest()` использует `TestOK()` (листинг 14.5).

Листинг 14.5. `TestOK()`

```

1: bool TestOK
2: {
3:     const char theOperator,
4:     const float theOperand,

```



```

5:      const float theExpectedResult
6:  )
7:  {    // Результат с плавающей точкой = Сумматор
8:      float Result = Accumulator(theOperator,theOperand);
9:
10:     if (Result == theExpectedResult)
11:     {
12:         cout << theOperator << theOperand <<
12:↵         " - succeeded." << endl;
13:         return true;
14:     }
15:     else // неудача
16:     {
17:         cout <<
18:             theOperator << theOperand << " - failed.<<
19:             "Expected " << theExpectedResult <<
19:↵             ", got " << result << // результат
20:             endl;
21:
22:         return false;
23:     };
24: }

```

Это — функция, которая выдает результат типа `bool` и имеет три неизменяемых параметра: оператор, операнд и результат, ожидаемый от сумматора. Функция выполняет операцию и сообщает, действительно ли результат равен тому, что ожидается.

Испытательные функции, подобные этой и `SelfTest()`, должны быть простыми. Вы должны быть способны проверить правильность испытательных функций с первого взгляда (т.е. путем *сквозного контроля*). Если они неправильны, вы будете охотиться за ошибками, которые не существуют или пропустите допущенные ошибки.

Небольшое изменение в ленте `Tape()`

Чтобы облегчить проведение испытаний, необходимо изменить только одну строку в функции ленты `Tape()`.

Замените

```
*3: static const int myTapeChunk = 20;
```

на

```
*3: static const int myTapeChunk = 3;
```

Почему? Чтобы проверить, что распределение динамической памяти работает, при испытании необходимо побудить

Таре () изменять размеры ленты, по крайней мере, однажды. Чтобы сделать это, размер участка нужно сделать меньше, чем число операций в испытании.

Выполнение программы

Теперь пришло время искать ошибки. Давайте выполним самопроверку:

ВЫВОД

```
1: !  
2:  
3: 0  
4: @0? succeeded.  
5: +3? succeeded.  
6: -2? succeeded.  
7: *4? failed. Expected 4, got 1  
8: Test failed.
```

При испытании программа потерпела неудачу!

Из строки 1 протокола испытаний видно, что был введен оператор самопроверки и нажата клавиша Enter. В строке 3 показано значение сумматора, сохраненное к началу испытания в строке 3 в SelfTest(). Испытания выполняются успешно, пока не выводится строка 7.

Обратите внимание, что выполнялись только три из четырех испытаний. Почему тест деления не был выполнен вообще?

Сокращение вычислений при вычислении операндов логического оператора &&

Логический оператор && обладает тем свойством, что если какой-то один его операнд является ложным, последующие выражения даже не будут вычисляться. Да и зачем их вычислять? Как только какой-нибудь операнд ложен, все выражение будет ложно! Это можно назвать *сокращением вычислений*.

Поэтому дальнейшие испытания не выполняются, если в каком-либо из них был обнаружен сбой, потому что ожидаемый результат каждого испытания требует правильного результата предшествующего испытания. Но, конечно, такая организация испытаний может доставить много хлопот, если вы хотите вычислить (выполнить) функцию в таком выражении независимо от результата предшествующих подвыражений.

Что было неправильно?

К счастью, самопроверка сразу указывает на строку с ошибкой — неправильно обрабатывается случай '*':

```
19:          case '*':                               // Случай '*':
20:
21:          myAccumulator = myAccumulator,theOperand;
```

Вы, конечно, обратили внимание, что программа не выполняет умножение в строке 21. Вместо * здесь стоит ,. Эту ошибку я не придумал специально. Я сделал эту ошибку при записи примера и обнаружил ее при самопроверке.

Почему компилятор не обнаружил эту ошибку? Оказывается, что *запятая* , является *инфиксным оператором*. Он возвращает значение крайнего справа операнда в качестве своего результата. Иногда, хотя и очень редко, этим удобно пользоваться.

Устранение ошибки и повторный запуск

Устранив ошибку и повторно запустив программу, вы увидите, что программа работает прекрасно. Кажется, даже нет никаких проблем с перераспределением памяти для ленты в программе Tape(). Но относитесь к этому скептически. Ошибки распределения динамической памяти могут скрываться даже в самом, казалось бы, безукоризненном коде и не обнаруживаться при испытаниях.

Отладка без отладчика

Если вы имеете интегрированную среду разработки (Integrated Development Environment — IDE), то у вас есть отладчик — специальная программа, которая позволяет видеть выполняемые строки программы и исследовать вычисляемые значения. Отладчик может даже перехватывать исключения и показывать вам строки, вызвавшие исключения. Но если вы не имеете отладчика, стратегии, обсуждаемые в следующих разделах, могут помочь найти причины ошибок во время выполнения.

Как поймать волка методом деления пополам (дихотомии) его территории обитания

Вообразите фермера, у которого на ферме каждую ночь воет волк. Чтобы поймать волка, фермер строит неприступный забор вокруг фермы, а также делит ферму забором пополам и затем слушает вой. Конечно, вой будет слышен с одной или с другой стороны забора. Тогда ту часть, с которой слышен вой, фермер делит пополам другим забором и затем слушает снова. Он повторяет процесс, пока не огородит забором столь небольшую территорию, что поймать на ней волка не составит труда.

Иногда ошибки, дающие о себе знать только во время выполнения, походят на волка. Они обнаруживают себя, но иногда трудно узнать точно, где же они находятся на самом деле. Определите всю область, в которой могут скрываться проблемы, и при входе и выходе из этой области распечатывайте значения критических переменных, которые указывают на то, что именно идет неправильно, — вполне возможно, что содержимое переменных подскажет вам это. Если значения правильные при входе и ошибочные при выходе, отобразите значения тех же самых переменных где-нибудь посередине. Повторяйте процесс деления, пока не найдете ошибку.

Печать значений

Вы использовали объекты `cout` и `cerr` для отображения значений. Позже вы научитесь выводить данные в файл на диске — это может быть еще лучшим методом поиска сложных ошибок.

Включение и отключение отладки с помощью команд `#define`

Команды `#ifdef`, `#define` и `#endif` позволяют избегать включения заголовка более одного раза. Вы можете также использовать команды препроцессора `#define` и `#ifdef` (новая команда, которая означает “если определен”) и `#endif`, чтобы компилировать некоторый код только тогда, когда определен некоторый конкретный символ. Например:

```
#define DEBUGGING
...
#ifdef DEBUGGING
cout << "Starting debug - value = << SomeValue << endl;
#endif
```

В этом коде символ `DEBUGGING` (ОТЛАДКА) определен (возможно, в начале программы, или даже во включаемом файле). Если этот символ определен, инструкция `cout` будет откомпилирована, и программа отобразит `SomeValue`.

Это удобно потому, что отладочные строки после `#ifdef DEBUGGING` не будут компилироваться и, естественно, не будут выполняться, если вы удалите `#define`.

Это уменьшает риск повреждения программы, так как для отключения инструкций отладки их не приходится удалять редактором.

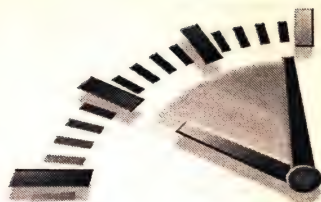
Резюме

На примере калькулятора вы научились использовать небольшой язык в качестве интерфейса пользователя. Кроме того, вы научились встраивать средства самопроверки, а значит, сумеете найти ошибку, обнаруженную средствами самопроверки. Вы также освоили некоторые стратегии отладки программ, пригодные даже тогда, когда среди доступных инструментальных средств вы имеете только компилятор и редактор.

УРОК 15

Структуры

И ТИПЫ



В этом уроке вы научитесь создавать наборы связанных друг с другом поименованных констант, структуры данных, в которых сможете хранить много переменных различных типов подобно тому, как в одном контейнере можно хранить самые разнородные предметы, а также освоите вызов функций с помощью указателей.

Организация разработки программ

К настоящему времени вы, вероятно, почувствовали ритм разработки программ: добавление возможностей, разработка функций, реализующих возможности, улучшение программы (рефакторинг — переразложение на классы), повторное испытание и повторение цикла с самого начала. Пришло время еще раз продемонстрировать все это на нашем примере.

На рис. 15.1 показана новая организация калькулятора. Сумматор `Accumulator()`, лента `Tape()` и части главной программы `main()`, которые взаимодействуют с пользователем, были перемещены в свои собственные модули. Все эти три модуля находятся в пространстве имен `SAMSCalculator`.

Стрелки показывают, какие функции вызывают друг друга: главная программа `main()` вызывает `CalculateFromInput()`, которая вызывает сумматор `Accumulator()` и ленту `Tape()`. Главная программа `main()` может также вызывать функции из других модулей.

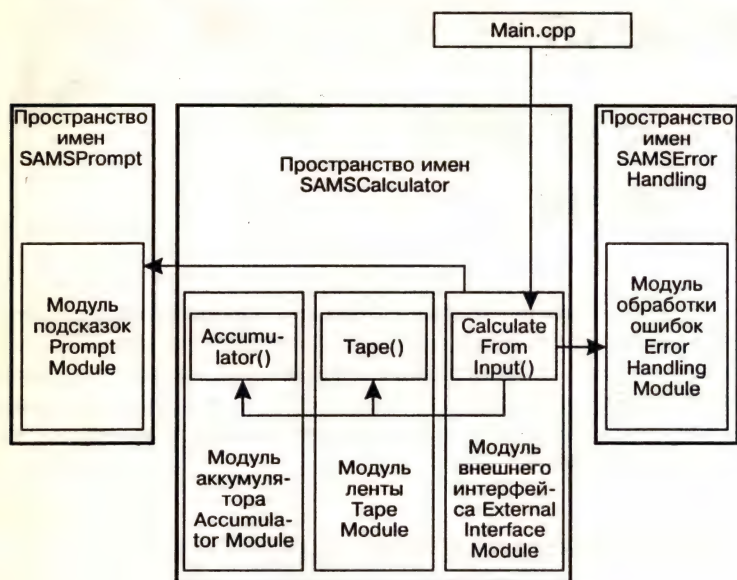


Рис. 15.1. Новая организация модулей

После завершения этой реорганизации пришло время использовать одну очень мощную возможность C++: возможность создавать новые типы данных.

До сих пор вы использовали “встроенные” типы вроде `char` (символ), `int` и `float` (вещественное число с плавающей точкой). Каждый из этих типов имеет некоторые ограничения, например, `char` (символ) может использоваться только для символов, `int` — только для чисел без дробных частей, `float` — только для вещественных чисел с плавающей точкой (фактически для любых вещественных чисел). Если вы создадите новый тип, компилятор будет следить за правильным использованием переменных этого типа и сгенерирует сообщение об ошибке, если переменная этого типа будет использоваться неправильно.

Объявление перечислимых типов

Перечислимый тип — тип данных, чьи переменные могут принимать значения только из набора констант, перечисленных в объявлении типа (*перечислимый* значит “перечисляемый (называемый) один за другим”). Например, вы можете объявлять перечислимый тип `anOperator` и предложить шесть возможных значений для него: `add` (добавить), `subtract` (вычесть), `multiply` (умножить), `divide` (делить), `reset` (сбросить, переустановить) и `query` (запрос) (они называются точно так же, как операторы сумматора `Accumulator()`).

Синтаксис для перечислимых типов:

```
enum{имя_константы, имя_константы, ...};
```

Вот пример перечисления:

```
enum anOperator {add,subtract,multiply,divide,reset,query};
```

Эта инструкция имеет две цели.

1. Она делает `anOperator` названием (именем) нового типа и ограничивает набор его значений шестью указанными значениями, аналогично тому, как `int` ограничен возможностью иметь в качестве значений только целые числа. Вы можете позже определить переменные типа `anOperator`.
2. Она делает `add` (добавить) символической постоянной со значением 0, `subtract` (вычесть) — символической постоянной со значением 1, `multiply` (умножить) — символической постоянной со значением 2, `divide` (делить) — символической постоянной со значением 3, `reset` (сбросить, переустановить) — символической постоянной со значением 4 и `query` (запрос) — символической постоянной со значением 5. Каждая символическая постоянная в перечислимом типе имеет в качестве значения число. Если какого-либо иного определения не предусмотрено, каждая постоянная по определению имеет в качестве значения номер своего положения в списке минус 1 (это точно так же, как индекс элемента массива: индекс первого элемента равен 0, второго — 1 и т.д.).

Вы можете теперь определять и использовать переменные нового типа:

```
1: enum anOperator
{add,subtract,multiply,divide,reset,query};
2: anOperator Operator;
3: Operator = '+'; // компилятор сгенерирует сообщение
                  // об ошибке
4: Operator = add; // компилируется успешно
```

Анализ

Строка 1 — объявление нового типа. Название (имя) типа имеет префикс “an”. Префикс “a” или “an” указывает на то, что тип определен пользователем, причем идентификатор не является стандартным типом, локальной переменной или константой, формальным параметром или статической локальной переменной.

В строке 2 определена переменная этого нового типа. В строке 3 компилятор обнаружит ошибку, потому что в этой строке программист пытается присвоить переменной значение, отличное от одной из перечисленных констант, зато строка 4 компилируется успешно, потому что в ней переменной присваивается перечисленная константа.

Использование перечислений в сумматоре Accumulator()

В листинге 15.1 показан заголовочный файл AccumulatorModule, в котором объявлено перечисление.

Листинг 15.1. Перечисление в заголовочном файле AccumulatorModule

```
1: #ifndef AccumulatorModuleH
2: #define AccumulatorModuleH
3:
4: namespace SAMSCalculator
5: {
6:     enum anOperator // Перечисление
7:     {add,subtract,multiply,divide,reset,query};
8:
9:     float Accumulator // Сумматор с плавающей точкой
10:     (
11:         const anOperator theOperator,
12:         const float theOperand = 0 // константа
13:     );
```

```

14: };
15:
16: #endif

```

Анализ

В строках 6 и 7 объявлен перечислимый тип (и так как он находится в заголовочном файле данного модуля, в любом пользователе данного модуля теперь можно определять переменные этого типа). Строка 11 указывает, что сумматору `Accumulator()` требуется передать параметр `theOperator` перечислимого типа — больше вы не можете передавать символ. Это делает сумматор `Accumulator()` более безопасным, потому что компилятор защитит его от ошибочных операторов.

Реализация изменилась незначительно, за исключением того, что все, имеющее отношение к ленте `Tape()`, было перемещено в функцию `ExternalInterfaceModule`, и теперь в этом модуле для случаев-меток в переключателе (инструкция `switch`) используются константы перечисления. Модифицированная программа показана в листинге 15.2.

Листинг 15.2. Перечисления в реализации `AccumulatorModule`

```

1: #include <exception>
2: #include <ios>
3:
4: #include "AccumulatorModule.h"
5:
6: namespace SAMSCalculator // пространство имен
7: {
8:     using namespace std; // станд. пространство имен
9:
10:    float Accumulator // Сумматор
11:    (
12:        const anOperator theOperator,
13:        const float theOperand
14:    )
15:    {
16:        static float myAccumulator = 0;
17:
18:        switch (theOperator) // переключатель (выбор)
19:        {
20:            case add: // случай(добавить):
21:                myAccumulator = myAccumulator +
22:                               theOperand;
23:                break;

```

```
23:
24:         case subtract:           // случай(вычесть):
25:             myAccumulator = myAccumulator -
                                   theOperand;
26:             break;
27:
28:         case multiply:           // случай(умножить):
29:             myAccumulator = myAccumulator *
                                   theOperand;
30:             break;
31:
32:         case divide:             // случай(делить):
33:             myAccumulator = myAccumulator /
                                   theOperand;
34:             break;
35:
36:         case reset:              // случай(сброс):
37:             myAccumulator = theOperand;
38:             break;
39:
40:         case query:              // случай(запрос):
41:             // Мы всегда возвращаем результат - не делать ничего
42:             break;
43:
44:         default:
45:             throw
*46:                 runtime_error      // Ошибка
47:                 ("Error - Invalid operator");
48:     };
49:
50:     return myAccumulator;
51: };
52: };
```

Обратите внимание, что строка 44 содержит заданный по умолчанию случай (метка default). Эта строка, вероятно, теперь не будет выполняться вообще, потому что компилятор проверяет правильность theOperand. Но если бы вы поддерживали эту программу и случайно удалили, скажем, строку 40, управление могло бы передано на заданный по умолчанию случай (на метку default) и вы получили бы уведомление об ошибке. Поэтому всегда лучше оставить заданный по умолчанию случай (метку default), даже если вы уверены, что она вам никогда не понадобится.

Объявление структурных типов

Несомненно, перечислимые типы полезны, но есть и более мощные определяемые пользователем типы, называемые *структурными типами*.

Структурные типы позволяют объединить набор *полей* в единственную переменную. В ней можно сохранять связанные данные вместе в единственном пакете и передавать его так же, как вы передаете `int` или `char`. Вы можете даже сделать массив структур.

Структурный тип объявляется с помощью ключевого слова `struct`, за которым следует название (имя) и любое количество членов с их типами:

```
1: struct имя_типа
2: {
3:     тип имя_переменной-члена . . .
4: };
```

Вот пример:

```
1: struct aTapeElement
2: {
3:     char Operator; // Символ Оператор
4:     float Operand; // Операнд с плавающей точкой
5: }
```

Структуры в стеке

Вы можете объявить, что переменная структурного типа должна размещаться в стеке. Например:

```
aTapeElement TapeElement;
```

Но как ее инициализировать? Для этого в C++ предусмотрен оператор выбора элемента (`.`). Он позволяет получить или установить значения поля:

```
TapeElement.Operator = '+'; // Оператор
TapeElement.Operand = 234; // Операнд
char Operator = TapeElement.Operator; // символ Оператор
```

Хорошо подобранное имя переменной структуры облегчает чтение инструкций.

Также возможно создать массивы структур:

```
aTapeElement TapeElement [20];
```

Вот как можно выбрать поля из любого элемента:

```
TapeElement[5].Operator = '+';  
TapeElement[5].Operand = 234;
```

Структуры в динамической памяти

Структуры также часто создаются оператором `new` (новый, создать). Например:

```
aTapeElement *TapeElement = new aTapeElement; // новый
```

Это можно прочитать так: “определить указатель на `aTapeElement` по имени `TapeElement` и инициализировать адресом выделенного с помощью оператора `new` в динамической памяти пространства, причем объем выделенной памяти должен быть равен размеру `aTapeElement`”.

Чтобы выбрать поля, можно использовать точку, но указатель нужно сначала разыменовать:

```
(*TapeElement).Operator = '+';  
(*TapeElement).Operand = 234;
```

Поскольку необходимость в этом возникает очень часто, в C++ есть стенографическая запись для этого выражения — оператор выбора элемента через указатель (`->`).

```
TapeElement->Operator = '+';  
TapeElement->Operand = 234;
```

Естественно, структуру из динамической памяти нужно обязательно не забыть удалить, причем делается это для структурных типов таким же образом, как и для простых типов:

```
delete TapeElement; // удалить
```

Вот как можно создать массив структур в динамической памяти:

```
aTapeElement *ATapeElement = new TapeElement[10];
```

Конечно, даже для массива структур в динамической памяти можно использовать оператор выбора элемента обычным способом:

```
TapeElement[5].Operand = 234;
```

И удаляется массив структур из динамической памяти также обычным способом:

```
delete [] TapeElement; // удалить
```

Однонаправленный связный список со структурами для ленты

Одна из интересных возможностей, возникающих при размещении структур в динамической памяти, заключается в том, что вы можете использовать указатель в структуре для того, чтобы связать все структуры в один список (называемый *связным списком*). Например, вы можете создать сколь угодно длинный список структур `aTapeElement`, прибавив к `aTapeElement` поле, в котором хранится указатель на следующий элемент `aTapeElement`:

```

1: struct aTapeElement
2: {
3:     char Operator; // Символ Оператор
4:     float Operand; // Операнд с плавающей точкой
5:     aTapeElement *NextElement;
6: };

```

Теперь структуры можно связать вместе:

```

1: aTapeElement Tape; // Лента
2: aTapeElement SecondElement = new aTapeElement;
                                   // новый элемент
3: Tape.NextElement = SecondElement; // Лента
4: aTapeElement ThirdElement = new aTapeElement;
                                   // новый элемент
5: Tape.NextElement->NextElement = ThirdElement;
                                   // Лента

```

В строке 5 использован оператор выбора элемента с помощью указателя, чтобы получить доступ к члену `NextElement` второго элемента и записать туда `ThirdElement`.

Давайте посмотрим, как все это можно использовать для ленты в нашей обновленной программе `Tare()`, которая приведена в листинге 15.3.

Листинг 15.3. Структуры в Type ()

```
1: void Tape // Лента
2: (const char theOperator, const float theOperand)
3: {
4:     static aTapeElement *TapeRoot = NULL;
        // ПУСТОЙ УКАЗАТЕЛЬ
```



```

5:
6:     if (theOperator == '?') // Печатать ленту
7:     {
8:         PrintTape(TapeRoot);
9:     }
10:    else if (theOperator == '.') // Программа
                                   //останавливается
11:    {
12:        DeleteTape(TapeRoot);
13:    }
14:    else // Нормальное действие: Добавить к ленте
15:    {
16:        aTapeElement *NewElement = new aTapeElement;
17:
18:        NewElement->Operator = theOperator;
19:        NewElement->Operand = theOperand;
20:        NewElement->NextElement = NULL;
21:
22:        if (TapeRoot == NULL) // если ПУСТОЙ
                                   // УКАЗАТЕЛЬ
23:        {
24:            // Это - первый Элемент
25:            TapeRoot = NewElement;
26:        }
27:        else
28:        {
29:            // Добавить элемент в конец после
30:            // последнего элемента в списке
31:
32:            // Начать с начала...
33:            aTapeElement *CurrentTapeElement =
                TapeRoot;
34:
35:            // ... пропустить до конца
36:            while
37:            (
38:                // НЕ ПУСТОЙ УКАЗАТЕЛЬ
                CurrentTapeElement->NextElement !=
                NULL
39:            )
40:            {
41:                CurrentTapeElement =
42:                    CurrentTapeElement->NextElement;
43:            };
44:
45:            // CurrentTapeElement - последний элемент
46:            // Добавить после него...
47:            CurrentTapeElement->NextElement =
                NewElement;
48:        };
49:    };
50: };

```

Анализ

Строки 16–47 добавляют новые записи к ленте. Логика функции стала более простой. Вы не должны делать перераспределение массива и копирование. Фактически больше не нужно проверять, заполнена ли лента, потому что всегда можно прибавить новый элемент, и у ленты нет никакого предела.

В строке 16 создается новый элемент ленты. Строки 18–20 присваивают значения его переменным-членам, для чего используется селектор члена в виде указателя. Обратите внимание, что член `NextElement`, который обычно указывает на следующий элемент, установлен равным `NULL` (**ПУСТОЙ УКАЗАТЕЛЬ**), что означает, что он “указывает на ничто”.

Строка 22 выясняет, имеются ли уже какие-либо элементы на ленте — когда указатель `TapeRoot` **НУЛЕВОЙ (ПУСТОЙ)** — `NULL`, новый элемент будет первым и его адрес будет присвоен `TapeRoot`.

Строки 33–43 выполняются, если на ленте уже есть один элемент (или большее количество элементов). Цикл просматривает список и останавливается на последнем элементе, причем этот элемент является единственным, у которого `NextElement` равен `NULL` (**ПУСТОЙ УКАЗАТЕЛЬ**).

В строке 47 указатель `NextElement` последнего элемента устанавливается так, чтобы указывать на новый элемент. Теперь новый элемент — последний.

Указатели на функции и обратные вызовы

Данные в стеке имеют местоположение — оно часто называется *адресом*. То же самое относится и к данным в динамической памяти. И то же самое относится и к каждой строке программы. На самом деле, когда вы читаете о передаче управления, фактически речь о переменной, которая операционной системой используется для того, чтобы следить, какая команда программы выполняется. Передача управления от начала к последующим командам есть не что иное, как перемещение по массиву с помощью индекса или указателя.

Фактически, программа, которая выполняет программы, могла бы выглядеть примерно так, как показано в листинге 15.4.

Листинг 15.4. Схема программы, выполняющей программы

```

1: struct anInstruction           // инструкция
2: {
3:     anInstructionCode InstructionCode;
4:     anInstruction *NextInstruction;
5: };
6:
7: anInstruction *Program =
8: new anInstruction[NumberOfInstructions];
9:
10: anInstruction *CurrentInstruction = Program;
11:
12: do // Делать
13: {
14:     Perform(CurrentInstruction.InstructionCode);
15:
16:     CurrentInstruction =
17:         CurrentInstruction.NextInstruction;
18: }
19: while (CurrentInstruction != NULL); // пока не ПУСТОЙ
                                     // УКАЗАТЕЛЬ

```

Поскольку `anInstruction` содержит указатель на следующую команду, эта программа может даже обрабатывать эквиваленты условных операторов и вызовов функций. На рис. 15.2 показана передача управления, которая выполняется точно так же, как при вызове функции в C++.

C++ позволяет получить адрес функции и использовать его для вызова функции очень похожим способом — с помощью *указателя на функцию*. Для указателей на функции, подобно другим указателям, выполняется контроль типов. Вот некоторые примеры их объявления:

```

typedef char (*ToGetAnOperator) (void);
typedef float (*ToGetAnOperand) (void);
typedef void (*ToHandleResults) (const float theResult);
typedef void (*ToDisplayMessage) (const char *theMessage);

```

В отличие от других объявлений типа, объявления указателя на функции требуют ключевого слова `typedef`. Другое (и последнее) различие между объявлением указателя на функцию и прототипом функции состоит в том, что название (имя) функции заменяется на `(*имя_типа)`. Символ `*` означает “указатель на функцию типа, имя которого есть `имя_типа`”.

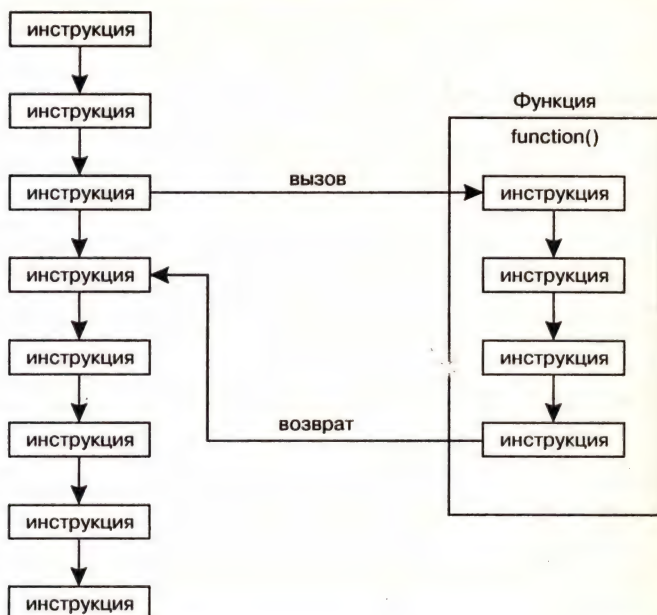


Рис. 15.2. Передача управления с помощью указателей

Вы можете объявлять переменные этих типов. Взгляните на объявление структуры с членами — указателями на функции:

```

1: struct aCalculatorExternalInterface
2: {
3:     ToGetAnOperator GetAnOperator;
4:     ToGetAnOperand GetAnOperand;
5:     ToHandleResults HandleResults;
6:     ToDisplayMessage DisplayMessage;
7: };

```

Но как же присваивать значения этим членам? Это показано в новой главной программе `main.cpp`, приведенной в листинге 15.5.

Листинг 15.5. Присвоения указателей на функции в `main.cpp`

```

1: char GetOperator(void)
2: {
3:     char Operator; // Оператор
4:     cin >> Operator; // Оператор
5:

```

```

6:      return Operator; // Оператор
7: }
8:
9: float GetOperand(void)
10: {
11:     float Operand; // Операнд
12:     cin >> Operand; // Операнд
13:
14:     return Operand; // Операнд
15: }
16:
17: void DisplayValueOnConsole(float theValue)
18: {
19:     cout << endl << theValue << endl;
20: }
21:
22: void DisplayMessageOnConsole(const char *theMessage)
23: {
24:     cout << theMessage << endl;
25: }
26:
27: int main(int argc, char* argv[])
28: {
29:     SAMSCalculator::aCalculatorExternalInterface
30:         CalculatorExternalInterface;
31:
32:     CalculatorExternalInterface.GetAnOperator =
33:         GetOperator;
34:
35:     CalculatorExternalInterface.GetAnOperand =
36:         GetOperand;
37:
38:     CalculatorExternalInterface.HandleResults =
39:         DisplayValueOnConsole;
40:
41:     CalculatorExternalInterface.DisplayMessage =
42:         DisplayMessageOnConsole;
43:
44:     return SAMSCalculator::CalculateFromInput
45:         (CalculatorExternalInterface);
46: }

```

Анализ

В строках 32–42 присваиваются адреса функций полям-указателям на функции. Конечно, это присваивание, а не вызов, потому что после имен функций нет круглых скобок или параметров.

Вызов функции с помощью указателя

Эта структура (и ее указатели на функции) используется для того, чтобы создать специальный вид образца программирования, известный как *обратный вызов*. Обратный вызов происходит, когда один модуль передает другому модулю указатели на некоторые из его функций так, чтобы другой модуль мог вызывать эти функции. В нашем случае обратный вызов используется для того, чтобы функция `CalculateFromInput()` могла использовать функции ввода и вывода главной программы `main.cpp` без необходимости знать что-нибудь о главной программе `main.cpp`.

Давайте рассмотрим функцию `NextCalculation()` в листинге 15.6. `NextCalculation()` вызывается `CalculateFromInput()` для того, чтобы реализовать цикл, который ранее был ядром главной программы `main.cpp`.

Листинг 15.6. `NextCalculation()`: вызов функций с помощью указателей

```

1: bool NextCalculation
2: (
3:     const aCalculatorExternalInterface // константа
4:     &theCalculatorExternalInterface
5: )
6: {
7:     char Operator = // Оператор
8:         theCalculatorExternalInterface.GetAnOperator();
9:
10:    switch (Operator) // переключатель (выбор)
11:        // (Оператор)
12:    {
13:        case '.': // случай '.': Остановка
14:        {
15:            return true;
16:        };
17:
18:        case '?': // случай '?': Распечатать ленту
19:        {
20:            Tape(Operator); // Лента (Оператор)
21:            return false;
22:        };
23:    }
24:
25:    // текущее значение, самотестирование и сброс
26:    case '=': case '@': // случай '=': случай '@':
27:    {

```



```

24:         anOperator OperatorValue =
25:             Operator == '=' ? query :
26:             reset;
27:         // Результат = Сумматор
28:         float Result = Accumulator(OperatorValue);
29:
30:         if (OperatorValue == query) // вопрос
31:         {
32:             theCalculatorExternalInterface.
32: *32: HandleResults(Result);
33:         };
34:
35:         return false;
36:     };
37:     // случай '+': случай '-': случай '*': случай '/':
38:     case '+': case '-': case '*': case '/':
39:     {
40:         float Number =
40: *40: theCalculatorExternalInterface.
40: *40: GetAnOperand();
41:
42:         anOperator OperatorValue =
43:             Operator == '+' ? add : // Оператор
44:                                     // == '+'? добавить:
45:             Operator == '-' ? subtract : // Оператор
46:                                     // == '-'? вычесть:
47:             Operator == '*' ? multiply : // Оператор
48:                                     // == '*'? умножить:
49:             divide; // делить
50:
51:         Accumulator(OperatorValue, Number);
52:                                     // Сумматор
53:         Tape(Operator, Number); // Лента
54:                                     // (Оператор, Число);
55:
56:         return false;
57:     };
58:
59:     case '!':
60:         SelfTest();
61:         return false;
62:
63:     // Что-нибудь другое - ошибка
64:     default:
65:     {
66:         throw runtime_error // Ошибка
67:             ("Error - Invalid operator.");
68:     };
69: };
70: };
71: }

```

Анализ

В строке 6 показан вызов функции `GetAnOperator()` из `main.cpp` с помощью указателя (на функцию) `GetAnOperator`.

В строке 32 показан вызов функции `DisplayValueOnConsole()` с помощью указателя `HandleResults`. А в строке 40 показан вызов функции `GetOperand()` из `main.cpp` с помощью указателя `GetAnOperand`.

Обратите внимание, что эти строки разорваны на операторе выбора элемента (`.`). Хотя компилятор позволяет это, обычно лучше не разрывать строки таким образом.

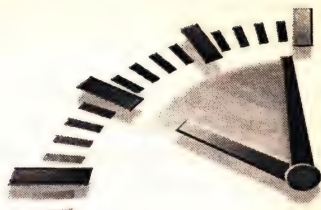
Обратите внимание также на то, что формальный параметр функции `NextCalculation` является ссылкой на структуру типа `aCalculatorExternalInterface`. Вы должны всегда передавать структуры по ссылке. Если вы отступите от этого правила, программа сделает копию фактического параметра (что отнимает много времени) и передаст ее функции (это также отнимает много времени), причем все изменения функция сделает в копии структуры, а не в самой структуре.

Резюме

Перечислимые типы, структурные типы и типы указателей на функции предлагают огромную мощь для создания продвинутых программ. Вы научились объявлять их, определять переменные новых типов, использовать константы перечислений, выбирать члены структур, а также присваивать значения указателям на функции и с их помощью вызывать сами функции. Все это поможет подготовиться к изучению объектно-ориентированных классов. Но сначала сделаем перерыв в изучении этого концептуально трудного материала и рассмотрим чтение и запись файлов на диск.

Урок 16

Файловый ВВОД-ВЫВОД



В этом уроке вы узнаете о чтении файлов с дисков и записи файлов на диски, а также научитесь записывать состояние программы и восстанавливать его.

Сохранение ленты между сессиями

До сих пор калькулятор выполнял все действия в ответ на ввод данных, и когда вы останавливали калькулятор, все введенные в него данные терялись. Это происходило потому, что переменные программы сохраняют свое содержание только во время выполнения программы. Вся совокупность их значений часто называется *состоянием* программы.

Большинство компьютеров имеет файловую систему, чье предназначение состоит в том, чтобы хранить данные между прогонами программ. И в каждом языке программирования предусмотрен способ поместить данные в файлы, находящиеся в файловой системе, и получать данные из них. C++ использует специальный вид потока — `fstream` (file stream — файловый поток), который подобен `cin` и `cout`, но операции ввода-вывода выполняются с файлами.

Файловые потоки `fstream`

В отличие от `cin` и `cout`, файловый поток `#include <fstream>` не имеет переменных, представляющих файлы. Вместо этого в программе необходимо определить переменную файлового потока типа `ifstream` (input file stream — входной файловый поток) или `ofstream` (output file stream —

выходной файловый поток) и затем открыть ее с помощью функции `fstreamvariable.open()`. (Здесь `fstreamvariable` обозначает идентификатор переменной файлового потока.) Чтобы открыть файл, в качестве параметра нужно указать путь к файлу в файловой системе операционной системы. Правильные способы определения таких путей должны быть описаны в документации операционной системы.

Если указать только имя файла, опуская другую информацию, то обычно предполагается, что файл находится там же, где и программа.



Файловая система

В отличие от большинства компонентов программы на C++, потоки ввода-вывода могут зависеть от операционной системы целевой машины, потому что в различных операционных системах пути к файлам определяются по-разному. Например, путь к файлу, который используется под Windows, может не подойти для Linux.

В идеале пользователи должны были бы вводить пути к файлу во время выполнения или во время установки программы на их систему, или же пути к файлу должны были бы вести в текущий каталог (для этого обычно не требуется ничего, кроме названия (имени) файла). Но даже при ссылке на файл в текущем каталоге использование расширений, столь широко распространенных в Windows, может не поддерживаться, и потому может вызвать проблемы.

Некоторые операционные системы поддерживают ссылки на файлы (псевдонимы файлов). Обращайтесь к документации операционной системы и помните о возможных проблемах, обусловленных использованием особенностей операционной системы.

Вы можете использовать переменную входного или выходного потока таким же образом, как `cin` и `cout`. Используйте оператор извлечения (`>>`) и оператор вставки (`<<`) для получения или запоминания данных.

По окончании работы поток нужно закрыть с помощью `close()`, хотя если переменная потока объявлена в стеке, поток будет закрыт автоматически, когда он выйдет из области видимости.



Область видимости

Область (или время) существования переменной. Большинство областей видимости переменных существуют лишь в течение того времени, пока управление находится в пределах того блока, в котором переменная определена (не считая вызовов функций). Область видимости глобальных и статических переменных — вся программа, хотя они пригодны для использования (часто говорят *видимы*) только в пределах того модуля или функции, в которых они определены.

Запись ленты на диск в потоке

Чтобы сохранить работу, сделанную в течение сессии с калькулятором, естественнее всего начать с записи ленты в функции `Tape()`. Лента (или функция `Tape()`) содержит все введенные в калькулятор команды. Если их сохранить на диске в конце сессии и затем прочитать в начале следующей сессии, то прочитываемая информация может использоваться для восстановления всего того, что вы сделали ранее.

Открытие и закрытие файла ленты

Новая версия `TapeModule` (см. листинг 16.1) имеет функцию `StreamTape()`, которая открывает выходной поток ленты, записывает содержимое ленты в этот поток, а затем закрывает его.

Листинг 16.1. `StreamTape()` в `TapeModule`

```

1: void StreamTape
2: (
3:   const char *theTapeOutputStreamName, // константа
4:   aTapeElement *theTapeRoot
5: )
6: {
7:     // если ПУСТОЙ УКАЗАТЕЛЬ
8:     if ((theTapeOutputStreamName == NULL)
9:         || (theTapeRoot == NULL)) return;
10:
11:     ofstream TapeOutputStream;
12:
13:     try
14:     {

```



```

*13:      TapeOutputStream.exceptions
*14:      (TapeOutputStream.failbit);
15:
*16:      TapeOutputStream.open
*17:      (theTapeOutputStreamName, ios_base::out);
18:
*19:      aTapeElement *CurrentTapeElement =
                                theTapeRoot;
20:      // не ПУСТОЙ УКАЗАТЕЛЬ
*21:      while (CurrentTapeElement != NULL)
22:      {
*23:          TapeOutputStream <<
*24:          CurrentTapeElement->Operator <<
                                // Оператор
*25:          CurrentTapeElement->Operand;
                                // Операнд
26:
*27:          CurrentTapeElement =
*28:          CurrentTapeElement->NextElement;
29:      };
30:
*31:      TapeOutputStream.close();
32:  }
*33:  catch (ios_base::failure &IOError) // отказ
*34:  {
*35:      SAMSErrorHandling::HandleOutputStreamError
*36:      (TapeOutputStream, IOError);
*37:  };
38:  }

```

Анализ

Строка 7 защищает функцию в ситуации, когда имя файла не задано. Если имя файла не задано, лента не записывается в файл.

В строке 9 определяется переменная типа выходного файлового потока (ofstream).

В строках 13–14 и 33–37 предусмотрена защита программы от ошибок в файловом потоке — таких как, например, недействительное “имя файла” или любое другое исключение.

В строке 13 для вызова исключений предусмотрена переменная потока — все сделано точно так же, как и для cin в ErrorHandlerModule.

В строке 16 полученное имя используется для открытия потока. Чтобы открыть файл для вывода, в этой строке функции открытия open() передается константа перечислимого типа ios_base::out. Если открыть файл не удастся, будет вызвано исключение, которое будет перехвачено в программе

обработки особых ситуаций в строке 33. Когда поток открывается с помощью `ios_base::out`, все, что уже есть в файле, вытирается, так что благодаря этому программа начинает работать с пустым файлом для текущей ленты.

В строке 19 начинается цикл, который просмотрит ленту, начиная с корневого элемента. В строке 21 проверяется, имеются ли какие-либо элементы на ленте, и выполняется цикл до тех пор, пока все еще есть элементы.

В строках 23–25 для записи оператора и операнда в файл используется уже знакомый оператор вставки (`<<`).

В строке 27 осуществляется переход к следующему элементу ленты, и цикл будет повторяться до тех пор, пока наконец указатель `NextElement` не станет равным `NULL` (ПУСТОЙ УКАЗАТЕЛЬ).

В строке 31 поток закрывается. Однако эта строка может быть опущена. Когда управление передается вне блока, переменная потока выходит из области видимости, и в результате поток закрывается автоматически.

Как заставить работать `StreamTape()`

Функции ленты `Tape()` нужен новый параметр, чтобы она могла передавать имя файла функции `StreamTape()`. Вот ее новый прототип:

```
1: void Tape // Лента
2: (
3:     const char theOperator, // Символ
4:     const float theOperand = 0, // с плавающей точкой
5:     const char *theTapeOutputStreamName = NULL
6: ) // указатель на символ = ПУСТОЙ УКАЗАТЕЛЬ
```

Новый параметр (и его значение по умолчанию) указан последним. Поскольку новый параметр имеет значение по умолчанию, его не нужно будет указывать в каждом вызове функции `Tape()`, так что не придется изменять каждый вызов функции `Tape()`, а нужно будет изменить лишь тот, в котором указывается имя.

Внутри функции ленты `Tape()` функция `StreamTape()` вызывается как раз перед `DeleteTape()`.

Указание имени файла для ленты

Путь к файлу для ленты будет введен извне, но не через `cin`. Для ввода будут использованы параметры главной функции `main()`; `argv` содержит элемент для каждого из слов в командной строке. Нулевое слово — название (имя) программы. Так что первое слово может быть названием (именем) файла ленты. Чтобы передать имя файла функции `Tape()`, в главной функции `main()` понадобится строка:

```
SAMSCalculator::Tape('.', 0, argv[1]);
```

Обратите внимание, что `argv` означает “argument value” (“значение параметра”).

Выбор ленты

Если запустить программу и открыть файл ленты с результатами, то будет выведено все, что было введено во время предыдущей сессии. Следующая стадия разработки состоит в том, чтобы заставить калькулятор читать ленту при запуске и исполнять записанные на ней команды так, как если бы вы их печатали. Для этого изменения понадобятся только в главном модуле `main.cpp`.

Проигрываем ленту, чтобы восстановить состояние

Функции `GetOperator()` и `GetOperand()` в `main.cpp` могут сначала прочитать входную информацию с сохраненной ленты. Когда лента закончится, они смогут переключиться на считывание входного потока из `cin`. В листинге 16.2 показано, как это работает.

Листинг 16.2. Главный модуль `main.cpp`: чтение ленты из файла

```
1: #include <iostream>
2: #include <ios>
3: #include <fstream>
4:
5: #include "PromptModule.h"
```



```
6: #include "ErrorHandlingModule.h"
7: #include "AccumulatorModule.h"
8: #include "TapeModule.h"
9: #include "ExternalInterfaceModule.h"
10:
11: using namespace std; // пространство имен
12: // Глобальная переменная для обратного вызова
*13: ifstream TapeInputStream;
14:
15: char GetOperator(void)
16: {
17:     char Operator;
18:
19:     if
20:     (
*21:         TapeInputStream.is_open() &&
*22:         (!TapeInputStream.eof())
23:     )
24:     {
*25:         TapeInputStream >> Operator; // Оператор
26:     }
27:     else
28:     {
*29:         cin >> Operator; // Оператор
30:     };
31:
32:     return Operator; // Оператор
33: }
34:
35: float GetOperand(void)
36: {
37:     float Operand = 1; // Операнд = 1
38:
39:     if
40:     (
*41:         TapeInputStream.is_open() &&
*42:         (!TapeInputStream.eof())
*43:     )
44:     {
*45:         TapeInputStream >> Operand; // Операнд
*46:     }
*47:     else
48:     {
*49:         cin >> Operand; // Операнд
*50:     };
51:
52:     return Operand; // Операнд
53: }
54:
55: void DisplayValueOnConsole(float theValue)
56: {
```

```
57:     cout << endl << theValue << endl;
58: }
59:
60: int main(int argc, char* argv[])
61: {
62:     SAMSErrorHandling::Initialize(); // Инициализация
63:
*64:     if (argc > 1) // Название (имя) файла указано
*65:     {
*66:         try
*67:         {
*68:             TapeInputStream.exceptions(cin.failbit);
*69:             TapeInputStream.open(argv[1], ios_base::in);
*70:         }
*71:         catch (ios_base::failure &IOError) // отказ
*72:         { // Файл не существует
*73:             SAMSErrorHandling::HandleInputStreamError
*74:                 (TapeInputStream, IOError);
*75: // Поток не будет открыт
*76: // но биты отказа не будут установлены
*77:         };
*78:
*79:     }; // в противном случае поток существует
*80: // но закрыт и не может использоваться;
81:
82:     SAMSCalculator::aCalculatorExternalInterface
83:         CalculatorExternalInterface;
84:
85:     CalculatorExternalInterface.GetAnOperator =
86:         GetOperator;
87:
88:     CalculatorExternalInterface.GetAnOperand =
89:         GetOperand;
90:
91:     CalculatorExternalInterface.HandleResults =
92:         DisplayValueOnConsole;
93:
94:     int Result = SAMSCalculator::CalculateFromInput
           // Результат
           (CalculatorExternalInterface);
95:
96:
97: // Не оставлять файл открытым, потому что иначе Tape()
98: // не сможет направить текущую сессию в поток
*99:     TapeInputStream.close();
100:
101: // Записать в поток и удалить ленту
*102:     SAMSCalculator::Tape('.', 0, argv[1]);
103:
104:     return Result; // Результат
105: }
```

Анализ

В строке 13 определена глобальная (в своем модуле) переменная `TapeInputStream`, причем другие модули не могут использовать эту переменную. Так как эта переменная глобальная, передавать поток функции `CalculateFromInput()` в качестве параметра не придется, а это позволит значительно уменьшить количество изменений в модуле `main.cpp`. Поскольку `GetOperator()` и `GetOperand()` определены в `main.cpp`, они смогут использовать эту переменную, даже если их вызвать из `ExternalInterfaceModule`. Это важное свойство позволяет обратным вызовам изменить состояние их родного модуля и использовать его переменные и функции. Кстати, это служит одним из немногих примеров правильного применения глобальных переменных.

В строке 21, чтобы проверить, действительно ли был открыт `TapeInputStream`, используется логическая функция `fstreamvariable.is_open()`. В строке 22 с помощью логической функции `fstreamvariable.eof()` проверяется, действительно ли достигнут конец файла. Если входной поток не был открыт или достигнут конец в файле команд, управление передается строке 29, которая получает следующий оператор из `cin`, а не из файла. Это означает, что программа будет работать, даже если вы не укажете имени файла, если файл существует, но пуст, и даже если некоторые команды записаны в файл. Если входной поток открыт, но конец файла не достигнут, в строке 21 оператор вводится из `TapeInputStream`.

В строках 39–50 то же самое делается для операнда.

`TapeInputStream` открывается в строках 64–80.

В строке 64 проверяется число слов в командной строке (`argc` означает “argument count” (“счетчик аргументов”)), чтобы определить, указан ли путь к файлу. Если путь указан, то строка 69 открывает `TapeInputStream`, чтобы использовать указанный путь.

Строка 99 выполняется в том случае, если пользователь ввел оператор `.`, чтобы указать, что программа должна остановиться. Тогда `TapeInputStream` закрывается так, чтобы функция `Tape()` смогла открыть файл ленты для вывода.

В строке 102 для записи ленты в поток вызывается `Tape()` и затем лента удаляется (освобождается память, в которой хранилась информация, записанная на ленту). Эта строка

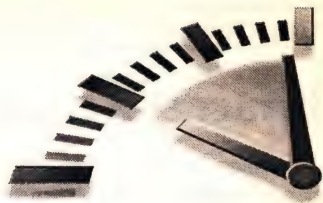
обычно находилась в `CalculateFromInput()`, но ее пришлось переместить сюда, чтобы не передавать `argv[1]` в `CalculateFromInput()`.

Резюме

Потребовалось относительно небольшое количество изменений, чтобы калькулятор помнил, что вы делали в прошлый раз, и возвращался к состоянию, достигнутому в прошлый раз. Для этого калькулятор использует файл ленты в качестве входного. Вы научились определять потоковые переменные, открывать их для вывода и ввода, а также закрывать их. Вы уже умеете использовать оператор вставки для записи в поток и оператор извлечения для чтения из потока. Итак, вы готовы приступить к созданию ваших первых объектов на C++.

УРОК 17

Классы: структуры с функциями



В этом уроке вы изучите понятие класса и узнаете, из каких компонентов состоят классы.

Класс как мини-программа

Усвоив материал предыдущих уроков, вы уже знаете почти все, что нужно для овладения концепцией *класса-типа*. Вы научились определять и использовать функции и структуры, содержащие разнообразные данные. Теперь вы изучите *классы*. В сущности, классы являются структурами, которые содержат как функции-члены, так и поля. Таким образом, классы могут хранить и обрабатывать данные.

Переменная, типом которой является класс, обычно называется *объектом*.

Так же, как внутреннее состояние программы или функции определяется содержимым ее переменных, внутреннее состояние объекта определяется содержимым его полей. Содержимое полей может изменяться при передаче управления функциям-членам объекта, причем объект сохранит эти изменения до следующего вызова функции-члена. Поля объекта могут быть простыми переменными, структурами, перечислениями или типами-классами.

Функции-члены ведут себя точно так же, как и любые другие функции, за исключением того, что они получают доступ к полям без помощи оператора выбора члена.

Классы и их экземпляры

Класс и объект класса — разные вещи, они отличаются друг от друга так же, как чертежи и здания. Класс описывает объекты, которые могут быть созданы с его помощью.

Создать объект — это значит создать *экземпляр* (образец) класса. Чтобы создать объект, нужно определить переменную, типом которой является класс. Это называется *созданием экземпляра* или *инстанцированием* (класса).

Поля (члены) объекта появляются тогда, когда объект создается, и автоматически разрушаются при разрушении объекта.

Объекты могут быть созданы в стеке или в динамической памяти, подобно любым другим переменным. Объект разрушается, когда программа заканчивается, когда он удаляется или когда его переменная выходит из области видимости.

Объявление класса

Объявление класса выглядит точно так же, как объявление структуры, за исключением того, что в нем используется ключевое слово `class` (класс):

```
class имя_класса // объявление класса
{
    члены
};
```

Некоторые классы имеют только поля (члены-данные), но чаще всего они имеют также и члены-функции.

Чтобы преобразовать сумматор в класс, нужно написать:

```
class aSAMSAccumulator // Класс
{
    float myAccumulator; // с плавающей точкой
    void Accumulate(char theOperator, float theOperand = 0);
};
```

Правда, здесь есть одна проблема. В отличие от структур, в которых члены обычно являются публичными, или общедоступными (`public`), т.е. доступными для вашей программы, элементы класса обычно частные, или приватные (`private`). Благодаря этому *скрывается информация*, что также называется *инкапсуляцией*, причем это является основным свойством класса. Хотя вы видите эти члены при чтении объявления, ни одна часть

программы не может использовать их. Так что ни `myAccumulator`, ни функция накопителя `Accumulate()` не доступны для вашей программы — все происходит так, как будто они были определены в части реализации модуля, а не в заголовке.

Чтобы сделать функцию накопителя `Accumulate()` *доступной* (или *публичной*), в объявлении класса нужно указать ключевое слово `public` (*общедоступный*, *публичный*). Это одно из нескольких ключевых слов видимости. Листинг 17.1 демонстрирует использование этого ключевого слова.

Листинг 17.1. Ключевое слово `public` (общедоступный, публичный)

```
1: class aSAMSAccumulator // Класс
2: {
3:     float myAccumulator; // с плавающей точкой
4:
5:     public: // общедоступный, публичный
6:
7:     void Accumulate
7:    (char theOperator, float
           theOperand = 0);
8: };
```

Ключевое слово `public`: действует до конца объявления класса или пока не будет указано другое ключевое слово видимости (например `private`: (частный:)).

Обычно лучше всего использовать ключевое слово `private`: (частный:), это гарантирует, что по чистой случайности члены, которые должны быть приватными, не окажутся общедоступными и наоборот. В листинге 17.2 показано использование этого ключевого слова.

Листинг 17.2. Ключевое слово `private`: (частный:)

```
1: class aSAMSAccumulator // Класс
2: {
3:     private: // Частный:
4:
5:     float myAccumulator; // с плавающей точкой
6:
7:     public:
8:
9:     void Accumulate
9:    (char theOperator, float theOperand = 0);
11: };
```

Заголовок и реализация

Заголовочный файл для класса содержит объявление класса, в котором объявляются функции-члены и поля. Файл реализации содержит определения функций-членов.

Поскольку класс в отношении сокрытия имен действует так же, как пространство имен, пространство имен обычно не используется в заголовке класса и файле реализации. Однако `#ifndef`, `#define` и `#endif` все же всегда включаются в заголовочный файл по обычным причинам.

Обычно модуль содержит объявление и определение только одного класса, но компилятор не сочтет ошибкой и наличие нескольких классов.

Каждая функция-член класса должна быть определена в файле реализации для модуля. Чтобы идентифицировать функцию как элемент класса, в заголовке функции перед названием (именем) функции указывается название (имя) класса:

```
тип имя_класса::имя_функции (параметры)
{
    тело
}
```

Оператор разрешения области видимости (`::`), который ранее использовался после имени пространства имен, указывает компилятору название (имя) класса, в котором функция объявлена, и позволяет компилятору удостовериться, что объявление соответствует определению.

Имя класса перед именем функции также позволяет в функции получить доступ к общедоступным и приватным членам класса без оператора выбора элемента.

Вызов функций-членов класса

Функции-члены вызываются извне объекта с помощью все того же самого оператора выбора элемента (`.`) и оператора выбора элемента с помощью указателя (`->`), которые обычно используются для обращения к полям структур (листинг 17.3).

Листинг 17.3. Операторы выбора элемента и выбора элемента с помощью указателя

```
1: SAMSAccumulator AccumulatorInstance; // экземпляр
                                     // сумматора
2:
3: aSAMSAccumulator *AccumulatorInstancePointer =
```

```

4:      new aSAMSAccumulator;
5:
*6: AccumulatorInstance.Accumulate('+',34);
*7: AccumulatorInstancePointer->Accumulate('-',34);

```

Типы, вложенные в классы

Поскольку класс часто играет роль пространства имен, связанные с ним типы могут быть объявлены в общедоступном разделе класса. Например, перечислимый тип `anOperator` может быть объявлен в классе `aSAMSAccumulator`, как показано в листинге 17.4.

Листинг 17.4. Перечисление в объявлении класса

```

1: class aSAMSAccumulator // Класс
2: {
3:     private: // Частный
4:
5:         float myAccumulator; // с плавающей
                               // точкой
6:
7:     public:
8:
9:         enum anOperator // Перечисление
*10:        {add , subtract , multiply , divide , query ,
        reset};
11: // {Прибавить, вычесть, умножить, разделить, запрос,
    // сбросить};
12:     void Accumulate
13:     (
14:         anOperator theOperator,
15:         float theOperand = 0 // с плавающей
                               // точкой
16:     );
17: };

```

Чтобы вне класса объявить переменную типа, вложенного в класс, используйте оператор разрешения области видимости так, как если бы этот тип был объявлен в пространстве имен:

```

aSAMSAccumulator Accumulator;
Accumulator.Accumulate(aSAMSAccumulator::add,34);

```

Конечно, это справедливо не только для вложенных перечислимых типов, но также и для вложенных структурных типов и типов-классов.

Однако для обращения к функциям-членам объекта оператор разрешения области видимости не применяется. Доступ к ним осуществляется в результате применения оператора выбора элемента к названию (имени) объекта.

Конструкторы и деструкторы

Когда вы создаете переменную-структуру, значение полей не определено, пока их не инициализирует программа. Но в классе можно объявить и определить специальную член-функцию, *конструктор*, который может инициализировать общедоступные и частные члены-переменные при создании объекта (экземпляра класса).

Конструктор вызывается автоматически кодом, который компилятор генерирует для создания объекта.

Конструктор имеет то же самое название (имя), что и класс. Он вовсе не имеет возвращаемого типа, недопустим даже `void`, причем самый основной конструктор не имеет никаких аргументов. Для класса `aSAMSAccumulator` он определен в декларации класса, как показано в листинге 17.5.

Листинг 17.5. Конструктор `aSAMSAccumulator`

```
1: class aSAMSAccumulator // класс
2: {
3:     private: // частный
4:
5:         float myAccumulator;
6:
7:     public: // общедоступный
8:
9:         enum anOperator
10:        {add,subtract,multiply,divide,query,reset};
11: // {Прибавить, вычесть, умножить, разделить, запрос,
    // сбросить};
*12:        aSAMSAccumulator(void);
13:
14:        void Accumulate
15:        (
16:            anOperator theOperator,
17:            float theOperand = 0
18:        );
19: };
```

Конструктор должен быть объявлен в общедоступном разделе объявления класса, обычно сразу после всех общедоступных типов и перед любыми общедоступными членами-функциями. Если конструктор частный, то экземпляр класса не может быть создан, и потому такие конструкторы применяются очень редко.

Раздел инициализации в конструкторе

Определение обычной функции и член-функции в файле реализации состоит из заголовка и тела, однако конструктор имеет дополнительный раздел между ними, специально предназначенный для определения инициализации членов-переменных. Например, следующий код инициализирует член-переменную `myAccumulator` нулем для каждого создаваемого экземпляра класса (объекта):

```
aSAMSAccumulator::aSAMSAccumulator (void): myAccumulator(0)
{
}
```

Вы можете также инициализировать поля в теле функции конструктора. Обычно это более приемлемо, если инициализация достаточно сложна и требует принятия некоторых решений или вызова функций, которые выполняются как часть инициализации.

Раздел инициализации состоит из названий (имен) полей, отделенных запятыми, причем после каждого имени переменной в круглых скобках указывается ее начальное значение. Эти начальные значения называются *инициализаторами*. В данном случае начальное значение — литерал.

Тело конструктора

Прототип конструктора объявляется в объявлении класса в заголовочном файле модуля, а тело конструктора определяется в файле реализации так же, как для любой член-функции. Не забудьте, что конструктор не возвращает никакого типа, в противном случае компилятор сгенерирует сообщение об ошибке.

Конструкторы с параметрами

Основной, или заданный *по умолчанию*, конструктор не имеет никаких параметров, но конструкторы могут иметь параметры. Объявление параметров конструктора делается точно так же, как объявление параметров функции-члена. Например:

```
aSAMSAccumulator (float theInitialAccumulatorValue);
```

Обычно конструктору передаются параметры, так что инициализацией члена-переменной можно управлять так, как показано ниже:

```
1: aSAMSAccumulator::aSAMSAccumulator
2:     (float theInitialAccumulatorValue):
3:     myAccumulator(theInitialAccumulatorValue);
4: {
5: }
```

Здесь инициализатор поля больше не является литералом. На сей раз это формальный параметр.

Позволяется также следующий вариант, но предыдущая форма определения предпочтительна:

```
1: aSAMSAccumulator::aSAMSAccumulator
1: { (float theInitialAccumulatorValue)
2: {
3:     myAccumulator = theInitialAccumulatorValue;
4: }
```

Множественные конструкторы

Итак, вы видели, что есть много видов конструкторов. Какой же вид конструктора наиболее подходящий? Как часто бывает в программировании, когда есть много альтернатив, приходится пользоваться ими всеми, выбирая наиболее подходящую для данного конкретного случая. Конечно, это же истинно и для конструкторов.

Класс может иметь несколько конструкторов (листинг 17.6), но компилятор вызовет тот, который создает экземпляр класса.

Листинг 17.6. Множественный конструктор в aSAMSAccumulator

```
1: class aSAMSAccumulator // Класс
2: {
3:     private: // частный
4:
5:     float myAccumulator; // с плавающей точкой
```



```

6:
7:     public:
8:
9:         enum anOperator // Перечисление
10:            {add, subtract, multiply, divide, query,
                                     reset};
11: // {Прибавить, вычесть, умножить, разделить, запрос,
    // сбросить};
*12:         aSAMSAccumulator(void);
*13:         aSAMSAccumulator
            (float theInitialAccumulatorValue);
14:
15:         void Accumulate
16:         {
17:             anOperator theOperator,
18:             float theOperand = 0 // с плавающей
                                     // точкой
19:         };
20: }

```

Благодаря наличию нескольких конструкторов при создании экземпляра класса может быть использована либо заданная по умолчанию инициализация `myAccumulator` значением 0, либо инициализация заданным для конструктора начальным значением, которое и будет присвоено `myAccumulator`:

```
aSAMSAccumulator FirstAccumulatorValueZero;
```

либо:

```
aSAMSAccumulator SecondAccumulatorValueSpecifiedAs(3);
```

Компилятор выберет конструктор на основании параметров их типов, которые указаны за названием (именем) переменной. Если параметров нет, используется конструктор, заданный по умолчанию.

Деструкторы

Мало того, что компилятор генерирует код, который вызывает выбранный вами конструктор при создании экземпляра класса (объекта), он также генерирует код, который вызывает другую специальную член-функцию, *деструктор*, когда объект выходит из области видимости или удаляется.

Деструктор имеет то же название (имя), что и класс, причем его имени предшествует "virtual ~" или "~" (знак ~ называется тильдой). В листинге 17.7 показан деструктор.

Листинг 17.7. Деструктор aSAMSAccumulator

```

1: class aSAMSAccumulator // класс
2: {
3:     private: // частный
4:
5:         float myAccumulator;
6:
7:     public: // общедоступный
8:
9:         enum anOperator
10:        {add,subtract,multiply,divide,query,reset};
11:
12:        aSAMSAccumulator(void);
13:        aSAMSAccumulator(float
                                theInitialAccumulatorValue);
14:
15:        virtual ~aSAMSAccumulator(void);
                                // виртуальный
16:
17:        void Accumulate
18:        (
19:            anOperator theOperator,
20:            float theOperand = 0
21:        );
22: };

```

В строке 15 показан деструктор этого класса. Есть ситуации, в которых ключевое слово `virtual` (виртуальный) не является необходимым, но всегда безопаснее указать его по причинам, которые станут понятны после изучения материала урока 22, “Наследование”.

Деструктор может быть только один, и он никогда не имеет ни возвращаемого типа, ни аргумента.

Обычно деструктор нужен лишь тогда, когда некоторые переменные-члены размещаются в куче. Объект с помощью деструктора должен удалить свои члены-переменные, размещенные в куче, чтобы предотвратить утечку памяти.

Конструктор копирования и его назначение

Есть одна специальная форма конструктора с аргументами: конструктор копии (листинг 17.8).

Он используется для создания объекта, состояние которого точно такое же, как и состояние некоторого другого объекта того же самого класса.

Листинг 17.8. Конструктор копии в `aSAMSAccumulator`

```

1: class aSAMSAccumulator
2: {
3:     private:
4:
5:         float myAccumulator;
6:
7:     public:
8:
9:         enum anOperator
9:  {add, subtract, multiply, divide, query, reset};
10:
11:         aSAMSAccumulator(void);
12:         aSAMSAccumulator(float
13:                             theInitialAccumulatorValue);
*13:        aSAMSAccumulator
13:  (aSAMSAccumulator theOtherAccumulator);
14:
15:         virtual ~aSAMSAccumulator(void);
16:
17:         void Accumulate
17:  (char theOperator, float theOperand = 0);
18: };

```

`aSAMSAccumulator` имеет конструктор копии, который может быть реализован следующим образом:

```

aSAMSAccumulator::aSAMSAccumulator
(aSAMSAccumulator theOtherAccumulator):
    myAccumulator(theOtherAccumulator.myAccumulator)
{
}

```

В этом определении интересно то, что используется частное поле `myAccumulator` из объекта `theOtherAccumulator`. Это допускается, потому что `theOtherAccumulator` — экземпляр того же самого класса, что и объект, чей конструктор копии вызывается.

Конструктор копии используется только при создании экземпляра объекта. Например:

```

aSAMSAccumulator SecondAccumulatorValueSpecifiedAs(3);

aSAMSAccumulator CopyOfSecondAccumulator
    (SecondAccumulatorValueSpecifiedAs);

```


В этом случае `CopyOfSecondAccumulator` инициализируется значением 3.

Ослабление правила “Объявить перед использованием” в классах

Члены класса объявляются в декларации класса. В реализации порядок деклараций в классе не влияет на видимость, так что член-функция, объявленная в классе первой, может обратиться к функции, объявленной пятой. В действительности, каждый член в декларации класса является видимым для каждой член-функции реализации.

Однако это ослабление правила не относится к объявлению класса. Приведенный ниже фрагмент содержит ошибку, потому что `anOperator` не был объявлен до его использования в формальном параметре функции `Accumulate()`:

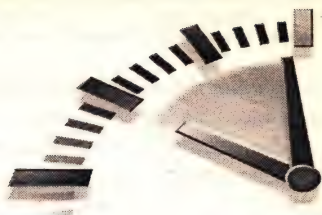
```
void Accumulate
(
    anOperator theOperator,
    float theOperand = 0
);
enum anOperator
{add, subtract, multiply, divide, query, reset};
```

Резюме

Вы узнали, что классы подобны структурам с функциями-членами. Вы получили представление об использовании ключевых слов `public` (общедоступный) и `private` (частный) для определения области видимости и узнали, что классы имеют специальные функции, называемые конструкторами и деструкторами, которые используются для инициализации и для освобождения динамической памяти, занимаемой данными, перед разрушением объектов. Вы узнали, что класс может иметь несколько конструкторов, включая и конструктор копии. Наконец, вы научились объявлять типы в классе и использовать их вне класса.

УРОК 18

Улучшение программы, или рефакторинг, — переразложение калькулятора на классы



В этом уроке вы разберете калькулятор и подготовитесь разложить его на классы.

Перенос функций в классы

Созданная версия калькулятора почти не отличается от объектно-ориентированной программы на C++. На этом уроке нам предстоит разработать документацию на объектно-ориентированный проект калькулятора, чтобы показать различия.

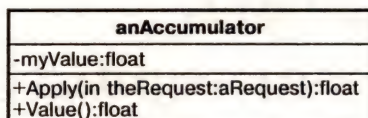
Диаграмма UML

Унифицированный язык моделирования (Unified Modeling Language, UML) — стандартный язык для рисования диаграмм объектно-ориентированных программ независимо от языка. Есть несколько типов диаграмм UML, но мы рассмотрим только один из них: *диаграммы класса*.

Диаграмма класса показывает классы, которые составляют программу, и их *отношения* (в UML их также называют

зависимостями) — особенно если один класс содержит поля другого класса. Имеется единственный блок для каждого класса, причем каждый блок разделен на разделы для имени класса, полей (называемых *атрибутами* в UML) и членов-функций (называемых *операциями* в UML).

На рис. 18.1 показан пример класса.



*Рис. 18.1. Диаграмма класса
в UML: блок-класс*

В этом примере показан класс `anAccumulator`. Он имеет поле `myValue`, в котором хранится число с плавающей точкой (`float`). Знак “минус” (-) перед именем члена-переменной указывает, что поле является частным (`private`). Тип поля указывается после названия (имени) поля, после двоеточия (:), которое разделяет имя и тип.

Класс также имеет две функции-члена: функцию `Apply()` (Применить), которая принимает неизменяемый (константа — `const`) параметр `aRequest` (называемый входным (слово `in`)) и возвращает число с плавающей точкой (`float`), и функцию `Value()` (Значение), которая не имеет параметров и возвращает число с плавающей точкой (`float`).

Перед именами обеих функций стоит знак “плюс” (+), который указывает, что они общедоступны (`public`).

Заданные по умолчанию конструкторы не показываются, хотя они могут быть. На диаграмме должны быть показаны все конструкторы с параметрами и деструктор.

Диаграмма UML для калькулятора

На рис. 18.2 показано, как калькулятор можно представить в виде набора классов.

Это — UML-диаграмма классов для всего пространства имен `SAMSCalculator`, которое является единственным пространством имен, используемым в объектно-ориентированной версии калькулятора.

SAMSCalculator

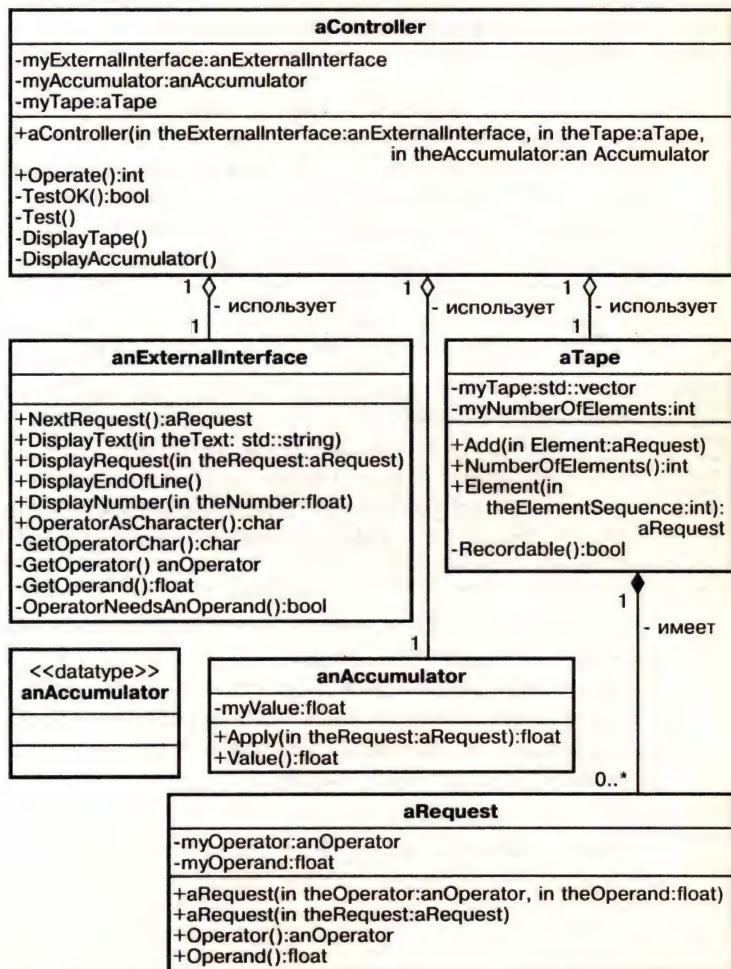


Рис. 18.2. Калькулятор как UML-диаграмма класса

На ней пять классов и тип данных UML (тип данных UML — любой тип, не являющийся классом). Этот тип данных UML — знакомый перечислимый тип `anOperator`. Не все, что будет обсуждаться далее, изображено на диаграмме, но может быть понято из названий (имен) типов и параметров.

Классами являются:

- `aController`, которому передаются `anExternalInterface`, `anAccumulator` и `aTape`. Внутри него содержатся ссылки на них, но он не является владельцем этих объектов (он не имеет их). Их времена жизни отличаются от времени жизни диспетчера, что показано связью от `aController` к *используемым* классам. Незаполненный ромб на соединяющих линиях показывает, что *совместно используемый* объект связанного класса “подключается” к родительскому объекту.



Время жизни объекта

Это время между созданием объекта и его разрушением. Объект создается либо когда он определяется как переменная, либо когда ему выделяется динамическая память. Объект разрушается, когда он выходит из области видимости или удаляется.

- `anExternalInterface`, который управляет всем вводом и выводом калькулятора. Контроллер использует его для всех операций ввода и вывода. `anExternalInterface` позволяет контроллеру использовать `GetRequest()`, который возвращает `aRequest`, причем он может также отображать `aRequest`, текст или число.
- `anAccumulator`, который `aController` использует, чтобы применить метод `Operand()` (Операнд) класса `aRequest` к `anAccumulator`. `Apply()` (Применить) также использует метод `Operand()` (Операнд) класса `aRequest`. Оба метода — `Apply()` (Применить) и `Value()` (Значение) — возвращают текущее `myValue` для `anAccumulator`.

- `aTape`, который `aController` использует для записи `aRequest`. `aTape` содержит `myTape`, вектор (`std::vector`) из `aRequest`, к которому функция `Add()` (Добавить) класса `aTape` прибавляет оператор и операнд каждой операции. Как и `iostream`, вектор `std::vector` — часть Стандартной библиотеки C++, он подобен тому, что вы использовали в предыдущей версии программы, когда реализовывали ленту как связный список `aTapeElement`.
- `aTape` позволяет использовать его `NumberOfElements()` и любой определенный элемент `Element()` при необходимости. `aTape` может иметь любое число объектов `aRequest`, причем он управляет существованием каждого из этих объектов; это показано закрашенным ромбом на соединительной линии, проведенной от `aTape` до `aRequest`, и обозначением `0..*` в ее конце возле `aRequest`.
- `aRequest`, который имеет поля (члены-переменные) `myOperator` и `myOperand`. `aRequest` передается каждому классу в программе и возвращается членами-функциями в двух из классов. Он имеет конструктор, который принимает `theOperator` и `theOperand`, и конструктор копии, который используется тогда, когда `aRequest` добавляется к `aTape`. Благодаря этому `aTape` может иметь его собственную копию `aRequest`, от которой он может избавиться, когда это потребуется.

Члены-функции `Operator()` и `Operand()` (Операнд) обеспечивают значения членов-переменных класса `aRequest` для других классов. Обратите внимание, что нет никакого способа изменить эти переменные после создания экземпляра класса (объекта), потому что эти переменные частные (`private`), а имеющиеся функции могут только получить их значения, но не устанавливать их.

Эта диаграмма может быть руководством при проектировании калькулятора как объектно-ориентированной программы.

Отличия

`PromptModule` и `ErrorHandlingModule` больше не существуют. Их функции будут помещены в `anExternalInterface` и в `aController`.

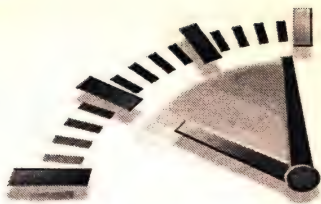
Чтобы облегчить чтение классов, некоторые названия (имена) были изменены. К ним относится название функции `Accumulate()`, которая теперь называется `Apply` (Применить) (так что теперь можно кодировать `Accumulator.Apply()` (Сумматор.Применить)), и внутреннее значение `anAccumulator`, которое теперь называется `myValue` и может быть получено через функцию `Value()` (Значение).

Следующий урок будет посвящен использованию `aTapElement`. Причем он получит более общее название — `aRequest`. Фактически теперь это основной объект калькулятора.

Резюме

Вы изучили унифицированный язык моделирования `Unified Modeling Language`, который выступает в качестве стандартной формы документации для объектно-ориентированных программ на любом языке. Вы научились читать основные схематические UML-изображения (диаграммы) классов и узнали, как на диаграммах изображаются классы, методы, параметры, возвращаемые типы, состояния `public` (общедоступное) и `private` (частное) и как изображается использование одними классами других и обладание одними классами экземпляров других классов в качестве своих членов. Вы рассмотрели различия между старой и новой программой, представленной на этой диаграмме. Эта диаграмма калькулятора пригодится, когда вы приступите к преобразованию калькулятора в объектно-ориентированную программу.

УРОК 19



Реализация калькулятора как системы классов

В этом уроке вы примените полученные знания о классах на практике.

Система обозначений класса

Как вы знаете, класс имеет две части: объявление класса, которое входит в заголовочный файл модуля, и определение класса, которое входит в файл реализации модуля.

Объявление простого класса, такого как `aRequest`, демонстрирует большую часть системы обозначений, используемой для определения класса (см. листинг 19.1).

Листинг 19.1. Заголовок `aRequest`

```
*1: #ifndef RequestModuleH
*2: #define RequestModuleH
3:
*4: namespace SAMSCalculator // пространство имен
*5: {
*6:     class aRequest // класс
*7:     {
*8:         public: // общедоступный
*9:
*10:         enum anOperator
*11:         {
*12:             add, // добавить
*13:             subtract, // вычесть
*14:             multiply, // умножить
```

```

*15:          divide, // делить
*16:          query, // запрос
*17:          reset, // сброс
*18:          selftest, // самотестирование
*19:          querytape,
*20:          stop // остановка
*21:      };
*22:
*23:  // Обратите внимание: нет конструктора по умолчанию
*24:  // вы должны указать оператор и операнд, когда
*25:  // создаете экземпляр (объект) этого класса
*26:
*27:      aRequest
*28:      ( // константы
*29:          const anOperator theOperator,
*30:          const float anOperand
*31:      );
*32:
*33:  // Позволить копирование
*34:
*35:      aRequest(const aRequest &theRequest);
*36:
*37:  // Нельзя изменить оператор или операнд;
*38:  // после создания экземпляра (объекта) этого класса
*39:  // можно только получить их значения
*40:
*41:      anOperator Operator(void) const;
*42:      float Operand(void) const;
*43:
*44:      private: // частный
*45:
*46:      anOperator myOperator;
*47:      float myOperand;
*48:  };
*49: };
*50:
*51: #endif

```

Анализ

Строки 1, 2 и 51 должны быть вам знакомы по заголовочным файлам старой версии. Они предотвращают сообщения компилятора типа

```
[C++ Error] RequestModule.h(11):
E2238 Multiple declaration for 'aRequest'
```

```
[Ошибка C++] RequestModule.h (11):
E2238 Множественное объявление для 'aRequest'
```

Строки 4, 5 и 49 заключают объявление класса в пространство имен, так что если в некоторой другой библиотеке

объявлен класс `aRequest`, он не будет иметь никакого отношения к нашему классу, хотя их названия (имена) совпадают. Иными словами, это предотвращает *коллизии имен*.

Строки 6, 7 и 48 составляют блок, в котором объявлен класс, это видно по ключевому слову `class` (класс).

В строках 10–21 объявлен перечислимый тип `anOperator` как часть этого класса.

Строки 27–31 и 35 — первичный конструктор и конструктор копии. Заданный по умолчанию конструктор не объявлен. Из-за этого любая попытка определить `aRequest` (запрос) вызвала бы генерацию компилятором следующего сообщения об ошибке:

```
[C++ Error] ControllerModule.cpp(20):
E2285 Could not find a match for 'aRequest::aRequest()'
```

```
[ Ошибка C++ ] ControllerModule.cpp (20):
E2285 Не могу найти соответствие для 'aRequest::aRequest()'
```

Отсутствие конструктора по умолчанию гарантирует, что при любом использовании `aRequest` будет правильно инициализирован значениями `theOperator` и `theOperand`.

Строки 46 и 47 — определения для `myOperator` и `myOperand`.

Строки 41 и 42 — функции чтения значений `myOperator` и `myOperand`. Они возвращают текущее значение этих членов и отмечены как `const` (константа). Это означает, что они не будут изменять какие-либо поля в классе.

Частные и общедоступные члены `aRequest`

Объявления `myOperator` и `myOperand` находятся в частном (`private`) разделе объявления класса (этот раздел начинается в строке 44). Поскольку они находятся в частном разделе, эти переменные не могут быть изменены чем бы то ни было, кроме функций-членов класса.

Поскольку конструктор устанавливает значения этих переменных, а функции-члены, которые обращаются к ним, только возвращают их значения, они не могут быть изменены после создания объекта.

Обычно поля должны быть частными (private). Тогда класс может выборочно предоставлять доступ как для чтения текущих значений (только через функции чтения значений), так и для изменения текущих значений без возможности их чтения (только через функции-механизмы установки), а также для чтения и изменения текущих значений (при наличии и функций чтения, и функций-механизмов установки).

Инициализация

В листинге 19.2 показано определение (реализация) класса aRequest.

Листинг 19.2. Реализация aRequest

```
*1: #include "RequestModule.h"
2:
*3: namespace SAMSCalculator // пространство имен
*4: {
*5:     aRequest::aRequest
*5:     (    // константы
*5:         const anOperator theOperator,
*5:         const float theOperand
*5:     ):
*6:     myOperator(theOperator),
*7:     myOperand(theOperand)
*8:     {
*9:     };
*10:
*11:     aRequest::aRequest(const aRequest &theRequest):
*12:     myOperator(theRequest.myOperator),
*13:     myOperand(theRequest.myOperand)
*14:     {
*15:     }
*16:
*17:     aRequest::anOperator aRequest::Operator(void) const
*18:     {
*19:         return myOperator;
*20:     };
*21:
*22:     float aRequest::Operand(void) const
*23:     {
*24:         return myOperand;
*25:     };
*26: };
```

Анализ

Строка 1 включает заголовочный файл с объявлением класса, чтобы компилятор мог проверить определение вместе с объявлением и удостовериться, что они совместимы друг с другом.

Строки 3 и 26 заключают функции в пространство имен, которое содержит объявление класса, чтобы компилятор проверил их совместимость с этим объявлением.

Строка 5 определяет первичный конструктор с его двумя параметрами. Это — один из двух способов инициализации экземпляра этого класса (второй — конструктор копии в строках 11–15). Строки 6 и 7 инициализируют члены-переменные `myOperator` и `myOperand` значениями `theOperator` и `theOperand`.

Вы должны указать имя класса, после которого следует оператор разрешения видимости перед каждым именем функции-члена, ибо в противном случае реализация функции не может быть связана с соответствующей член-функцией в объявлении класса.



Не забудьте указать имя_класса:: для каждой функции-члена

Пропуск конструкции `имя_класса::` является обычной ошибкой, причем компилятор не ловит ее, пока вы не начнете компоновать программу. При компоновке в таком случае обычно генерируется сообщение типа

```
[Linker Error] Unresolved external
'SAMSCalculator::aController::TestOK
(SAMSCalculator::anAccumulator&,
const SAMSCalculator::aRequest&, const float)
const'
from ObjectOrientedCalculator.cpp
```

[Компоновщик Ошибка] Неразрешенная внешняя ссылка

В строках 11–15 определяется конструктор копии, чей единственный параметр — ссылка на другой экземпляр того же самого класса. Инициализаторы в строках 12 и 13 устанавливают `myOperator` и `myOperand` равными `theRequest.myOperator` и `theRequest.myOperand`. Помните, что только конструктор копии может обращаться к полям в част-

ном разделе другого экземпляра, и что эти поля нельзя использовать для других способов изменения другого экземпляра класса.

В строках 17 и 22 в конце каждого заголовка функции-считыватели (получатели значений) отмечены как константы (`const`). Это значит, что они не будут изменять поля.

Внутреннее состояние

Внутреннее состояние `aRequest` представлено переменными `myOperator` и `myOperand`. Функции в строках 17–35 реализации позволяют другим классам или коду получать эти значения.

Если вы посмотрите на рис. 18.2 из урока 18, “Улучшение программы, или рефакторинг, — переразложение калькулятора на классы”, вы убедитесь, что не все классы имеют внутреннее состояние. Например, `anExternalInterface` вообще не имеет никаких полей.

Экземпляры таких классов, как `aRequest`, на протяжении всей своей жизни имеют единственное (одно и потому постоянное) внутреннее состояние, потому что их поля не могут быть изменены.

Экземпляры таких классов, как `anAccumulator`, часто изменяют свое внутреннее состояние. Каждый вызов `Apply()` (Применить) изменяет внутреннее состояние `anAccumulator`.

Экземпляры таких классов, как `aController`, имеют косвенное внутреннее состояние. Их поля не изменяются, но эти поля — объекты, которые имеют внутреннее состояние, и это состояние может изменяться через какое-то время.

Именно из-за возможности наличия внутреннего состояния столь важно объявлять функцию как константу (`const`). Если функция объявлена как константа (`const`), компилятор может проверить, что она не изменяет члены-переменные класса. Он может даже сгенерировать сообщения об ошибках, если функция, которой в качестве параметра-константы (`const`) передается экземпляр класса, вызывает любую не-константную член-функцию этого объекта. Так что ключевое слово `const` (константа) способствует тому, чтобы каждый объект использовался по назначению. Вы должны ис-

пользовать ключевое слово `const` (константа) настолько часто, насколько это возможно, — и для параметров, и для функций, чтобы уменьшить риск любого неправильного использования объекта.

В листинге 19.3 показана реализация класса `anAccumulator` — класса с более изменчивым внутренним состоянием, чем `aRequest`.

Листинг 19.3. Реализация `anAccumulator`

```

1: #include <string> // строки
2: #include <exception> // исключения
3:
4: #include "AccumulatorModule.h"
5:
6: namespace SAMSCalculator // пространство имен
7: {
8:     using namespace std; // используемое
                           // пространство имен
9:
10:    anAccumulator::anAccumulator(void): myValue(0)
11:    {
12:    };
13:
14:    anAccumulator::anAccumulator
15:        (anAccumulator &theAccumulator):
16:    myValue(theAccumulator.myValue)
17:    {
18:    };
19:
20:    float anAccumulator::Apply(const aRequest
                                &theRequest)
21:    { // переключатель (выбор Оператора)
22:        switch (theRequest.Operator())
23:        {
24:            case aRequest::add:
                                // случай добавить:
25:                myValue+= theRequest.Operand();
                                // += Операнд
26:                break;
27:
28:            case aRequest::subtract:
                                // случай вычесть:
29:                myValue-= theRequest.Operand();
                                // -= Операнд
30:                break;
31:
32:            case aRequest::multiply:
                                // случай умножить:

```

```

*33:             myValue*= theRequest.Operand();
                                // *= Операнд
*34:             break;
*35:
*36:             case aRequest::divide:
                                // случай делить:
*37:             myValue/= theRequest.Operand();();
                                // /= Операнд
*38:             break;
*39:
*40:             default:
*41:
*42:             throw
*43:                 runtime_error
*44:                 ( // строка
*45:                 string("SAMSCalculator::") +
*46:                 string("anAccumulator::") +
*47:                 string("Apply") + // Применяется
*48:                 string(" - Unknown operator.")
*49:                 ); // Известный оператор
*50:         };
*51:
*52:         return Value();
*53:     };
*54:
*55:     float anAccumulator::Value(void) const
*56:     {
*57:         return myValue;
*58:     };
*59: };

```

Анализ

В строке 10 реализации `myValue` инициализируется значением 0 для случая, когда заданным по умолчанию конструктором создается экземпляр `anAccumulator`.

Строки 14–16 — заголовок и инициализация для конструктора копии. Значение `myValue` для текущего экземпляра устанавливается равным значению `myValue` фактического параметра `theAccumulator`.

Когда завершается какой-либо конструктор, объект создан полностью, и поле `myValue` находится в безопасном и пригодном для использования состоянии. Если это поле не инициализировать, оно могло бы содержать любое случайное значение. Именно поэтому столь важны конструкторы.

По этой же причине в правильно написанных классах нет общедоступных (`public`) полей. Что касается общедоступных (`public`) полей, то отследить присвоение им значений невоз-

можно, и ничто не предотвращает присвоение несоответствующего значения в неподходящее время. Но код функции-члена имеет доступ и потому может проверять правильность входного значения, так что использование получателей и механизмов установки более безопасно с точки зрения сохранения непротиворечивости внутреннего состояния объекта в течение всей жизни объекта.

Строки 24–38 изменяют состояние поля `myValue`, когда вызов `Apply()` (Применить) делается для экземпляра `anAccumulator`. Чтобы сделать нужные изменения, в коде используется стенографическая запись арифметических операторов присвоения `+=`, `-=`, `*=` и `/=`.

В строках 40–49 учитывается возможность получения непредвиденного оператора, в этом случае вызывается исключение `runtime_error`. Обратите внимание, что при обработке исключения сообщается пространство имен, класс и функция, в которых обнаружена ошибка, а также выводится текст сообщения. Это может помочь при отладке.

Также обратите внимание на использование класса строк Стандартной библиотеки C++ (команда `#include <string>` в заголовке, пространство имен `std`), чтобы сгенерировать сообщение при вызове этого исключения. Класс строк `std::string` позволяет соединить несколько строк в одну с помощью знака `+`, аналогично тому, как использование нескольких операторов `<<` `cout` позволяет вывести несколько переменных в одну строку вывода. `string()` (строка) — конструктор для класса строк, он конвертирует (преобразовывает) строки-литералы в стиле C++ (фактически данные типа `char *` (символы)) в строки `std::string`.

Соглашения об именовании

Несмотря на применение классов и объектно-ориентированного подхода, названия вещей изменились лишь незначительно. Как и прежде, именам типов предшествуют префиксы “a” или “an”, а именам формальных параметров — префикс “the”. Это облегчает определение области видимости, источника, а также продолжительности жизни переменных и их содержимого. По-

мимо этого, такое соглашение позволяет отличить объявление класса от экземпляра класса.

Кроме того, продолжает использоваться префикс “my” (“мой”), хотя и в слегка отличном контексте.

В процедурном варианте функции `Accumulator()` (Сумматор), локальная переменная `myAccumulator` была статической, и потому продолжительность ее жизни была та же самая, что и у программы. Благодаря этому сумматор (формально реализованный в виде функции `Accumulator()`) имел внутреннее состояние и более походил на объект. В классах префикс “my” (“мой”) используется для полей и указывает по существу то же самое обстоятельство: “Данная переменная — часть внутреннего состояния объекта, она является *чем-то моим* собственным”.

Единственное различие состоит в том, что теперь продолжительность жизни этой переменной такая же, как и продолжительность жизни объекта, а не как продолжительность жизни программы.

Кроме того, каждый вызов сумматора (функции `Accumulator()`) воздействовал на ту же самую переменную `myAccumulator`. Это все еще справедливо в том смысле, что каждая программа, вызывающая `Apply()` (Применить) к тому же самому экземпляру `anAccumulator`, воздействует на ту же самую `myValue`; но как вы увидите в функции `SelfTest()`, теперь можно иметь несколько экземпляров `anAccumulator`, каждый с его собственной отдельной переменной `myValue`.

Некоторые программисты следуют другим стандартам именования. Некоторые делают название типа частью имени переменной, другие первую букву имени класса делают прописной, чтобы отличить класс от экземпляра. Однако в этой книге используется соглашение об именованиях, в котором основное внимание уделяется тому, что является самым важным с точки зрения владения объектами в объектно-ориентированной программе, источника содержимого переменных, продолжительности жизни объектов и различий между классами, экземплярами классов и формальными аргументами.

Перемещение кода функций в член-функции

В реализации сумматора Accumulator код функции сумматора Accumulator() перемещен в функцию Apply() (Применить) с минимальными изменениями. Теперь давайте рассмотрим заголовок и файл реализации aController, который воплощает большую часть того, что обычно находилось в функции калькулятора Calculator() (см. листинги 19.4 и 19.5).

Листинг 19.4. Заголовок aController

```

1: #ifndef ControllerModuleH
2: #define ControllerModuleH
3:
4: #include "ExternalInterfaceModule.h"
5: #include "AccumulatorModule.h"
6: #include "TapeModule.h"
7:
8: namespace SAMSCalculator // пространство имен
9: {
10:     class aController // класс
11:     {
12:     public: // общедоступный
13:
14:         aController
15:         (
16:             anExternalInterface
17:                 &theExternalInterface,
18:             anAccumulator &theAccumulator,
19:             aTape &theTape
20:         );
21:
22:         int Operate(void);
23:
24:     private: // частный
25:
26:         anExternalInterface &myExternalInterface;
27:         anAccumulator &myAccumulator;
28:         aTape &myTape;
29:
30:         bool TestOK // логический
31:         (
32:             anAccumulator &theAccumulator,
33:             const aRequest &theRequest,
34:             // константа
35:             const float theExpectedResult
36:             // константа

```



```

34:
35:         ) const; // константа
36:
37:     void SelfTest(void) const; // константа
38:     void DisplayAccumulator(void) const;
                                     // константа
39:     void DisplayTape(void) const; // константа
40: };
41: };
42:
43: #endif

```

Заголовок является простым, так что давайте начнем экспертную оценку реализации с рассмотрения конструктора в листинге 19.5.

Листинг 19.5. Конструктор aController

```

1: aController::aController
2: (
3:     anExternalInterface &theExternalInterface,
4:     anAccumulator &theAccumulator,
5:     aTape &theTape
6: ):
7:     myExternalInterface(theExternalInterface),
8:     myAccumulator(theAccumulator),
9:     myTape(theTape)
10: {
11: };

```

Анализ

Конструктор по умолчанию не задан, поэтому чтобы сделать aController, нужно использовать именно этот конструктор. Ему в качестве параметров нужно передать theExternalInterface, theAccumulator и theTape. Параметры — ссылки на экземпляры указанных классов. Как видно из строк 25–27 заголовка aController, инициализируемые поля также являются ссылками.

Обычно, как вы помните, компилятор не позволяет присваивать что-либо ссылке, если ссылка уже была определена. Однако в строках 25–27 объявления ссылки были только объявлены. Поэтому компилятор считает, что они не определены, пока управление не было передано телу конструктора. Поэтому эти ссылки могут быть инициализированы в разделе инициализации конструктора в строках реализации (16–18).

Теперь давайте рассмотрим изменения в функции SelfTest(), приведенной в листинге 19.6.

Листинг 19.6. Функция SelfTest() реализации aController

```

1: void aController::SelfTest(void) const
2: {
3:     anAccumulator TestAccumulator;
4:
5:     try
6:     {
7:         if
8:         (
9:             TestOK
10:            (
11:                TestAccumulator,
12:                aRequest(aRequest::add,3),
13:                                     // добавить
14:                3
15:            )
16:            &&
17:            TestOK
18:            (
19:                TestAccumulator,
20:                aRequest(aRequest::subtract,2),
21:                                     // вычесть
22:                1
23:            )
24:            &&
25:            TestOK
26:            (
27:                TestAccumulator,
28:                aRequest(aRequest::multiply,4),
29:                                     // умножить
30:                4
31:            )
32:            &&
33:            TestOK
34:            (
35:                TestAccumulator,
36:                aRequest(aRequest::divide,2),
37:                                     // делить
38:                2
39:            )
40:        { // Испытание прошло хорошо.
41:            cout << "Test OK." << endl;
42:        }
43:        else
44:        { // Испытание потерпело неудачу.
45:            cout << "Test failed." << endl;
46:        }
47:    }
48: }

```

```

45:     catch (...)
46:     { // Испытание потерпело неудачу
        // из-за исключения.
47:         cout << "Test failed because of an exception.";
48:     };
49: };

```

Анализ

В строке 3 показано одно из наиболее важных отличий от процедурной версии: эта функция `SelfTest()` создает экземпляр `anAccumulator`¹ и выполняет все испытания на этом объекте, а не на `myAccumulator`. Это позволяет избежать сброса текущего внутреннего состояния калькулятора и упрощает испытание. Функция `TestOK()` изменилась так, что может использовать этот объект, который передается как параметр. В строках 9–14 показан сам вызов.

`TestAccumulator` функции `SelfTest()` будет разрушен, когда `SelfTest()` будет завершена. Причем это никак не повлияет на экземпляр `myAccumulator` класса `aController`.

В строке 12 показан вызов конструктора `aRequest` в фактических параметрах `TestOK()` для создания временного запроса, используемого в процессе испытаний.

Теперь давайте рассмотрим, как используется одна из ссылок на конструктор `aController`, которая содержится в главной программе `main()` (см. листинг 19.7).

Листинг 19.7. Функция `DisplayTape()` класса `aController`

```

1: void aController::DisplayTape(void) const // константа
2: {
3:     int NumberOfElements = myTape.NumberOfElements();
4:
5:     for
5: {
5:         int Index = 0; // Индекс = 0
5:         Index < NumberOfElements;
// Индекс < NumberOfElements
5:         Index++ // Индекс ++

```

¹ Обратите внимание, что таким образом сжато выражен тот факт, что на самом деле создается экземпляр (`TestAccumulator`) класса `anAccumulator`. Эта возможность появилась благодаря использованию соглашения об именовании, так как в соответствии с этим соглашением, сразу видно, что `anAccumulator` — класс, а не объект, так как он имеет префикс “an”. Далее такие сокращения используются довольно часто. — Прим. ред.


```

5:  )
6:  {
7:      myExternalInterface.    // Элемент(Индекс)
7:  DisplayRequest(myTape.Element(Index));
8:      myExternalInterface.DisplayEndOfLine();
9:  };
10:
11:  DisplayAccumulator();
12: };

```

Анализ

В строке 7 функция `DisplayRequest()` объекта `myExternalInterface` используется для отображения запроса оператора с помощью `Operator()` и операнда с помощью `Operand()`.

Наконец, функция `Operate()` действительно управляет калькулятором, как показано в листинге 19.8.

Листинг 19.8. Функция `Operate()` класса `aController`

```

1: int aController::Operate(void)
2: {      // Запрос
*3:     aRequest Request =
*3:  myExternalInterface.NextRequest();
4:     // Запрос.Оператор() != aRequest::останов
*5:     while (Request.Operator() != aRequest::stop)
6:     {
7:         try
8:         { // переключатель -- выбор (Запрос.Оператор)
*9:             switch (Request.Operator())
10:            {
*11:                case aRequest::selftest:
12:
13:                    SelfTest();
14:                    break;
15:
*16:                case aRequest::querytape:
17:
18:                    DisplayTape();
19:                    break;
20:
*21:                case aRequest::query: // случай запрос:
22:
23:                    DisplayAccumulator();
24:                    break;
25:
26:                default:
27:
*28:                    myTape.Add(Request);

```

```

// Добавить Запрос
*29: myAccumulator.Apply(Request);
30: // Применить Запрос
31: };
32: // Запрос
33: Request = myExternalInterface.NextRequest();
34: }
*35: catch (runtime_error RuntimeError)
36: { // ошибка во время выполнения
37:     cerr << "Runtime error: " <<
37:  *   RuntimeError.what() << endl;
38: }
*39: catch (...)
40: { // Перехвачено исключение, отличное
    // от runtime_error
41:     cerr <<
42:  *   "Non runtime_error exception " <<
42:  *   "caught in:Controller.Operate." <<
43:     endl;
44: };
45: }; // Нет никаких фатальных ошибок,
    // в результате которых
46: // код возврата мог бы быть отличен от 0
47: return 0;
48: };

```

Анализ

Строка 3 получает `aRequest`² от `myExternalInterface`, а строка 5 повторяется в цикле, пока оператор (значение, возвращаемое функцией `Operator()`) отличен от останова (`aRequest::stop`).

В строке 9 показано, что в переключателе (инструкция `switch`) можно использовать перечисление (`enum`), вложенное в класс `aRequest`.

Строки 11, 16 и 21 — метки-случаи, получающие управление в зависимости от запрошенного оператора (`Request.Operator()`), проверяемого в строке 9.

В строках 28 и 29 `myTape` и `myAccumulator` используются для выполнения основных функций калькулятора.

И конечно, строки 35 и 39 перехватывают любые исключения, чтобы не прервать цикл.

² В соответствии с применяющимся соглашением об именовании, это означает, что на самом деле будет получен экземпляр класса `aRequest`. — *Прим. ред.*

Объект как структура обратного вызова

В уроке 15, “Структуры и типы”, вы изучили структуры с указателями на функции, которые в калькуляторе (точнее, в функции `Calculator()`) использовались для *обратного вызова* нужных функций ввода и вывода из главной функции `main.cpp`. Чтобы выполнить необходимые действия, `aController` получает набор объектов извне и координирует их взаимодействие, вызывая их функции-члены. Эти и подобные им образцы программирования показывают еще одну связь между концепциями процедурного и объектно-ориентированного программирования.

Кто выделяет память, кто удаляет, кто использует и что разделяется (используется совместно)

Вопрос о собственности объектов — один из самых важных в объектно-ориентированном программировании. В случае `theAccumulator`, `theExternalInterface` и `theTape`, передаваемых `aController`, экземпляры класса принадлежат той части программы, в которой определен экземпляр класса `aController` (в данном случае главной программе `main()`). `aController` не уполномочен избавиться от этих объектов, потому что он имеет только ссылки на них. Но компилятор “не навязывает такую политику”, так что в подобных случаях вы должны найти подходящее решение сами.

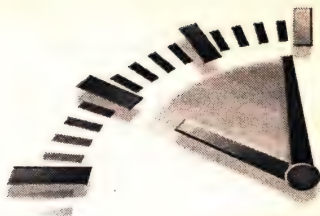
Другие классы, например `aRequest` и `anAccumulator`, являются полноправными владельцами памяти, отведенной их членам-переменным, которые создаются и разрушаются вместе с экземпляром класса.

Резюме

Вы рассмотрели несколько новых классов, которые являются частью новой версии калькулятора, и видели их объявления и определения. Их реализации подобны процедурным версиям, хотя и немного отличаются от них. Вы также видели различные способы реализации внутреннего состояния классов и различия в способах управления памятью объектами. Кроме того, вы научились передавать объект другому объекту, а затем обращаться к нему, чтобы выполнить определенные функции.

УРОК 20

Остальные классы калькулятора



В этом уроке вы изучите остальную часть реализации калькулятора.

Использование классов Стандартной библиотеки C++

Одно из главных отличий в работе программиста, применяющего объектно-ориентированные методы, состоит в изменении подхода к разработке. Если раньше программисты создавали все на пустом месте, то теперь, применяя объектно-ориентированные методы, программисты сначала ищут (возможно, уже созданные) классы, которые могут выполнить работу.

Очевидно, что такой подход может значительно повысить производительность труда программиста. Независимо от того, являетесь ли вы профессионалом или любителем, библиотеки классов не только экономят вам время, но и позволяют создать намного более мощные программы. Поверьте, что хорошая библиотека классов заменяет целый штат программистов мирового класса.

Стандартная библиотека классов C++ вполне подходит для того, чтобы начать поиск класса, с помощью которого можно достичь определенной цели и выяснить, было ли ранее написано что-нибудь подобное. Почти каждый компилятор имеет такую библиотеку в качестве стандартной опции.

Вы можете найти и другие библиотеки классов как свободно (т.е. бесплатно (freeware) или условно бесплатно (shareware)) распространяемое обеспечение в Internet. В проекте открытых каталогов DMOZ Open Directory Project приведены списки библиотек для PC и Macintosh — их можно найти 'по адресам http://dmoz.org/Netscape/Computing_and_Internet/Shareware/PC/Development_Tools/Components_%26_Libraries/ и http://dmoz.org/Netscape/Computing_and_Internet/Shareware/MAC/Development_Tools/Components_%26_Libraries/ соответственно.

Важно помнить, что каждый создатель библиотеки имеет различные идеи о хороших названиях (именах), о лучшей организации и интерфейсе классов, а также о том, как библиотека должна себя вести. Это означает, что иногда вы будете чувствовать себя так, как если бы вы собирали машину (или какой-нибудь другой механизм) из весьма разнообразных частей на кладбище старых автомобилей. Но это — просто заранее ожидаемая плата, не зависящая от вашего объектно-ориентированного языка, а выгода от повышения производительности столь велика, что перевешивает незначительные различия в эстетике и качестве, которые обусловлены библиотеками классов.

Использование библиотеки классов в aTape

Класс aTape намного меньше процедурной функции ленты Tape(). Частично это обусловлено тем, что код, сохраняющий ленту в файле, был удален. (Но немного позже вы убедитесь, что он будет добавлен снова, см. урок 22, "Наследование".) Другой фактор, способствующий уменьшению размера, — то, что теперь для хранения списка объектов aRequest будет использоваться класс vector (вектор) Стандартной библиотеки C++, и потому не нужно делать свой собственный связанный список структур aTapeElement.

Вектор vector — часть Стандартной библиотеки классов C++, одобренная ISO/ANSI. Поэтому применять ее безопасно, хотя код ее весьма отличается от того, который вам уже знаком. Но этот код хорошо проверен и последовательно следует внутренним стандартам именования (соглашениям об именовании).

В листингах 20.1 и 20.2 приведены заголовок и реализация `aTape`.

Листинг 20.1. Заголовок `aTape`

```

1: #ifndef TapeModuleH
2: #define TapeModuleH
3:
*4: #include <vector> // вектор
5:
6: #include "RequestModule.h"
7:
8: namespace SAMSCalculator // пространство имен
9: {
10:     class aTape // класс
11:     {
12:     public: // общедоступный
13:
*14:         aTape(void);
15:
16:         void Add(const aRequest &theRequest);
            // Добавить
17:         int NumberOfElements(void) const;
18:         // Возвращает копию
*19:         aRequest Element
20:             (const int theElementIndex) const;
21:
22:     private: // частный
23:
*24:         std::vector<aRequest> myTape;
            // станд.: вектор
25:
26:         bool Recordable
27:             (const aRequest &theRequest) const;
28:     };
29: };
30: #endif

```

Анализ

В строке 4 объявления класса включается `<vector>` (`<вектор>`), чтобы в строке 24 можно было создать объект-вектор `vector` в соответствии с его определением в пространстве имен `std`.

В строке 14 объявлен конструктор по умолчанию. Компилятор генерирует код для конструктора по умолчанию, даже если он не определен. Поэтому формально это не требуется, но лучше всегда объявлять конструкторы, которыми вы собираетесь пользоваться.

В строке 24 указывается, что вектор содержит только объекты а Request. Для этого используется специальная декларация, в которой тип объектов заключается в угловые скобки (<>).

Конструктор копии не объявлен для этого класса.



Сгенерированные компилятором конструкторы копии могут быть опасны

Если конструктор копии не является частью вашего класса и выполняется копирование объекта этого класса, то будет выполнено почленное копирование. Полученная копия может быть опасна, если хотя бы некоторые ее члены представляют собой указатели на данные в куче. При почленном копировании копируются указатели, а не содержимое памяти по адресу, хранящемуся в указателе. Таким образом, скопированный экземпляр будет иметь указатели на ту же самую память для хранения элемента данных, что и объект-оригинал, и если один экземпляр изменяет то, что находится в этой памяти, оба объекта будут работать с измененным содержанием. Возможно, это будет именно то, что как раз и нужно, но чаще всего это ошибка, обнаружить которую порой очень трудно.

В нашем случае это не проблема, потому что класс-вектор vector правильно копирует свое содержимое в заданную по умолчанию почленную копию.

Листинг 20.2. Реализация аTape

```

1: #include <exception> // исключение
2:
3: #include "TapeModule.h"
4:
5: namespace SAMSCalculator // пространство имен
6: {
7:     using namespace std; // используем станд.
                          // пространство имен
8:
9:     aTape::aTape(void)
10:    {
11:    };
12:
13:    bool aTape::Recordable
14:    (const aRequest &theRequest) const
15:    {

```

```

*15:         return // true если
*16:         (           /* Оператор */
*17: /* добавить */ (theRequest.Operator() ==
                        aRequest::add) ||
*18: /* вычесть */ (theRequest.Operator() ==
                        aRequest::subtract) ||
*19: /* умножить */ (theRequest.Operator() ==
                        aRequest::multiply) ||
*20: /* делить */ (theRequest.Operator() ==
                        aRequest::divide) ||
*21: /* сброс */ (theRequest.Operator() ==
                        aRequest::reset)
*22:         );
*23:     };
24:
25:     void aTape::Add(const aRequest &theRequest)
                                /* Добавить */
26:     {
*27:         if (Recordable(theRequest))
*28:         {
*29:             myTape.push_back(theRequest);
*29: // Сделать копию запроса, добавить к концу
*30:         };
31:     };
32:
33:     int aTape::NumberOfElements(void) const
34:     {
*35:         return myTape.size();
36:     };
37:         // Элемент
*38:     aRequest aTape::Element
*38: // (const int theElementIndex) const
39:     {
*40:         if (theElementIndex < 0)
41:         {
42:             throw // Ошибка во время выполнения
                    // программы
43:             runtime_error
44:             ( // строка
45:             string("SAMSCalculator::aTape::") +
46:             string("Element") + // Элемент
47:             string(" - Requested element
                                before 0th.")
48:             ); // Получен запрос об элементе
                    // перед 0-ым.
49:         }
50:
*51:         if (theElementIndex >= NumberOfElements())
52:         {

```



```

53:             throw
54:             runtime_error
55:             ( // строка
56:             string("SAMSCalculator::aTape::") +
57:             string("OperatorAsCharacter") +
58:             string(" - Request for element
           past end.")
59:             ); // Получен запрос об элементе
           // после конца.
60:         };
61:
*62:         return myTape[theElementIndex];
63:     };
64: };

```

Анализ

Строки 7–9 реализации содержат пустое определение конструктора по умолчанию. Хотя он и ничего не делает, его нужно определить, поскольку он объявлен.

Строки 27–30 добавляют элемент к `myTape`, если условие `Recordable()` истинно (`true`). Функция `Recordable()`, определенная в строках 13–23, необходима потому, что некоторые запросы не должны записываться на ленту, к ним относятся запросы `aRequest::querytape` и `aRequest::query`. Функция добавления `myTape.Add()` запоминает `aRequest`¹, так что символьное представление `aRequest::anOperator` не попадает в `aTape`². Этим новая версия отличается от старой функции ленты `Tape()`, которая хранила символ оператора в `aTapeElement`.

Строка 29 добавляет запрос к вектору `myTape`. Имя функции-члена выбрано в соответствии с необычным для вас соглашением, которое, тем не менее, является обычным в Стандартной библиотеке классов C++. В соответствии с этим новым соглашением, слова разделяются символами подчеркивания и все символы набираются на нижнем регистре.

Функция `push_back()` создает копию `theRequest` и добавляет ее к концу вектора. Это означает, что внутреннее со-

¹ Конечно, в соответствии с соглашением об именовании, это означает, что запоминается какой-то экземпляр класса `aRequest`. — *Прим. ред.*

² И опять, в соответствии с соглашением об именовании, это означает, что символ оператора не будет записан в экземпляр класса `aTape`. В дальнейшем подобные разъяснения не делаются. — *Прим. ред.*

стояние `aTare` не подвергается опасности, когда объект, представленный ссылкой `theRequest`, выходит из области видимости и разрушается. Если бы в этом списке хранились ссылки или указатели, то эти ссылки или указатели могли бы в любое время работать с памятью, которая была освобождена, т.е. возвращена в динамическую память.

В строке 38 определен получатель (считыватель) элемента — функция `Element()`. Строки 40 и 51 гарантируют, что вызывающая программа не может запрашивать элемент, чей последовательный номер меньше нуля или больше количества элементов в векторе (имеющего тип `vector`). Хотя вектор (имеющий тип `vector`), вероятно, вызвал бы исключение в такой ситуации, условие этой ошибки можно сначала проверить, а затем вызвать более информативное исключение. Это — один из примеров услуг, которые можно предоставлять в функциях-механизмах установки или в функциях-получателях. А если бы вы просто открыли `myTare` как общедоступную (`public`) переменную-член, то уровень риска был бы гораздо более высоким.

Интерфейс пользователя в объекте

Класс `anExternalInterface` “заботится” обо всем в интерфейсе пользователя. Однако в этой версии реализации код восстановления состояния калькулятора с помощью сохраненной ленты был удален. (Этот код будет снова представлен в уроке 22, “Наследование”).

Класс `anExternalInterface` теперь содержит все, что связывает калькулятор с внешним миром. Ни один из других классов (кроме `aController`, который всегда посылает результаты `SelfTest()` на `cout` или `cerr`) не связывается с внешним миром. В листинге 20.3 показан заголовочный файл `anExternalInterface`.

Листинг 20.3. Заголовок `anExternalInterface`

```
1: #ifndef ExternalInterfaceModuleH
2: #define ExternalInterfaceModuleH
3:
4: #include "RequestModule.h"
```

```

5:
6: namespace SAMSCalculator           // пространство имен
7: {
8:     class anExternalInterface       // класс
9:     {
10:         public:                     // общедоступный
11:
12:             anExternalInterface(void);
13:
14:             aRequest NextRequest(void) const;
15:
16:             void DisplayText(const char *theText)
17:                                     const;
18:             void DisplayRequest
19:                 (const aRequest &theRequest) const;
20:             void DisplayNumber(const float theNumber)
21:                 const;
22:             void DisplayEndOfLine(void) const;
23:
24:             char OperatorAsCharacter
25:                 (aRequest::anOperator theOperator)
26:                                     const;
27:
28:         private:                    // частный
29:
30:             char GetOperatorChar(void) const;
31:             aRequest::anOperator GetOperator(void)
32:                                     const;
33:
34:             bool OperatorNeedsAnOperand
35:                 (aRequest::anOperator theOperator)
36:                                     const;
37:
38:             float GetOperand(void) const;
39:
40:     };
41: };
42: #endif

```

Анализ

Обратите внимание на отсутствие внутреннего состояния. Никаких полей в этом классе нет.

В строке 21 введена специальная функция, которая превращает `aRequest::anOperator` обратно в его строковый эквивалент. Она используется для некоторых сообщений в `aController::TestOK()`, а также в самом классе.

Взгляните теперь на реализацию. Вы должны начать с конструктора (листинг 20.4), который теперь устанавливает `cin.exceptions` в строке 3.

Листинг 20.4. Конструктор `anExternalInterface`

```
1: anExternalInterface::anExternalInterface(void)
2: {
*3:     cin.exceptions(cin.failbit);
4: };
```

В листинге 20.5 показано, как получить оператор и вернуть его в качестве `aRequest::anOperator`.

Листинг 20.5. `anExternalInterface`. Получение `anOperator`

```
*1: char anExternalInterface::GetOperatorChar(void) const
*2: {
*3:     char OperatorChar;
*4:     cin >> OperatorChar;
*5:     return OperatorChar;
*6: };
7:
8: aRequest::anOperator anExternalInterface::GetOperator
9:     (void) const
10: {
*11:     char OperatorChar = GetOperatorChar();
12:
*13:     switch (OperatorChar) // переключатель (выбор
                            // OperatorChar)
14:     { // случай
*15:         case '+': return aRequest::add;
                            // '+': добавить
*16:         case '-': return aRequest::subtract;
                            // '-': вычесть
*17:         case '*': return aRequest::multiply;
                            // '*': умножить
*18:         case '/': return aRequest::divide;
                            // '/': делить
*19:         case '=': return aRequest::query; // '='
*20:         case '@': return aRequest::reset;
*21:         case '?': return aRequest::querytape;
*22:         case '!': return aRequest::selftest; // '!'
*23:         case '.': return aRequest::stop;
                            // '.': останов
24:
25:     default:
26:
*27:         char OperatorCharAsString[2];
```

```

*28:         OperatorCharAsString[0] = OperatorChar;
*29:         OperatorCharAsString[1] = '\0';
30:
*31:         throw // Неизвестный оператор
32:             runtime_error
33:             ( // строка
34:               string("SAMSCalculator::") +
35:               string("anExternalInterface::") +
36:               string("GetOperator") +
37:               string(" - Unknown operator: ") +
38:               string(OperatorCharAsString)
39:             );
40:     };
41: };

```

Анализ

Строки 1–6 представляют собой обычный код, предназначенный для получения символа оператора. Эта функция вызывается в строке 11, а ее результат используется в строке 13, чтобы в строках 16–23 транслировать символ к `aRequest:: anOperator`.

Строка 25 — заданный по умолчанию случай для недействительного оператора, а строки 27–29 преобразуют недействительный символ оператора в строку, которая используется исключением, вызываемым в строке 31.

Процесс получения операнда, показанный в листинге 20.6, аналогичен тому, который был в процедурной версии программы.

Листинг 20.6. `anExternalInterface`. Получение операнда
 Operand

```

*1: float anExternalInterface::GetOperand(void) const
*2: {
*3:     float Operand; // Операнд
*4:
*5:     try
*6:     {
*7:         cin >> Operand; // Операнд
*8:     }
*9:     catch (...)
*10:    {
*11: // Очистить состояние входного потока
*12:     cin.clear();
*13:
*14: // Избавиться от оставшихся ошибочных символов
*15:     char BadOperand[5];
*16:     cin >> BadOperand;

```

```

*17:
*18:         throw
*19:             runtime_error
*20:             ( // строка
*21:                 string("SAMSCalculator::") +
*22:                 string("anExternalInterface::") +
*23:                 string("GetOperand") +
*24:                 string(" - Not a number: ") +
*25:                     // Не число
*26:                 string(BadOperand)
*27:             );
*28:
*29:     return Operand; // Операнд
*30: };

```

Анализ

Строки 1–30 получают операнд и обрабатывают все ошибки ввода, вызывая исключение, которое пропускает ошибочные символы.

Как показано в листинге 20.7, функция `NextRequest()` выполняет ввод и упаковывает введенную информацию в `aRequest`.

Листинг 20.7. `anExternalInterface`. Получение запроса `aRequest`

```

1: bool anExternalInterface::OperatorNeedsAnOperand
2: (aRequest::anOperator theOperator) const
3: {
4:     return
5:     (
6:         (theOperator == aRequest::add) ||
7:         (theOperator == aRequest::subtract) ||
8:         (theOperator == aRequest::multiply) ||
9:         (theOperator == aRequest::divide) ||
10:        (theOperator == aRequest::reset)
11:    );
12: };
13:
14: aRequest anExternalInterface::NextRequest(void) const
15: {
16:     aRequest::anOperator Operator = GetOperator();
17:     // Оператор
18:     if (OperatorNeedsAnOperand(Operator))

```



```

// Оператор
19:    {
20:        return aRequest(Operator, GetOperand());
// Оператор
21:    }
22:    else
23:    {
24:        return aRequest(Operator, 0); // Оператор
25:    };
26: };

```

Анализ

В строках 1–12 определяется, нужен ли для конкретного оператора операнд.

Строки 14–26 управляют получением оператора и операнда и упаковкой их в объект `aRequest`.

Другие функции практически не изменились бы, если бы не появились новые функции, предназначенные для преобразования оператора из перечислимого типа в символ и отображения `aRequest` (листинг 20.8).

Листинг 20.8. `anExternalInterface`. Отображение запроса `aRequest`

```

*1: char anExternalInterface::OperatorAsCharacter
*2: (aRequest::anOperator theOperator) const
*3: {
*4:     switch (theOperator) // переключатель
// (выбор theOperator)
*5:     { // случай
*6:         case aRequest::add:         return '+';
// добавить: '+'
*7:         case aRequest::subtract:    return '-';
// вычесть: '-'
*8:         case aRequest::multiply:    return '*';
// умножить: '*'
*9:         case aRequest::divide:      return '/';
// разделить: '/'
*10:        case aRequest::query:       return '=';
*11:        case aRequest::reset:       return '@';
*12:        case aRequest::querytape:   return '?';
*13:        case aRequest::selftest:    return '!';
*14:
*15:        default:
*16:
*17:        throw
*18:        runtime_error // Неизвестный оператор
*19:        ( // строка

```

```

*20:             string("SAMSCalculator::
*20:↵             anExternalInterface::") +
*21:             string("OperatorAsCharacter") +
*21:↵             string(" - Unknown operator to be translated.")
*22:             );
*23:         };
*24:     };
25:
26: void anExternalInterface::DisplayRequest
26:↵     (const aRequest &theRequest) const
27: {
28:     cout <<                                     // Оператор
*29:         OperatorAsCharacter(theRequest.Operator()) <<
30:         theRequest.Operand(); // Операнд
31: };

```

Анализ

Строки 26–31 отображают запрос. Строка 29 обращается к функции `OperatorAsCharacter()`, чтобы получить символ оператора из запроса. Теперь только `anExternalInterface` знает, как конвертировать (преобразовать) символы в `aRequest::anOperator` и наоборот.

main.cpp

`main.cpp` (листинг 20.9) создает все необходимые экземпляры, передает объекты `aController`, а затем запускает калькулятор.

Листинг 20.9. main.cpp

```

1: #include "ExternalInterfaceModule.h"
2: #include "AccumulatorModule.h"
3: #include "TapeModule.h"
4: #include "ControllerModule.h"
5:
6: int main(int argc, char* argv[])
7: {
8:     SAMSCalculator::anExternalInterface
    ↵                                     ExternalInterface;
9:     SAMSCalculator::anAccumulator      Accumulator;
                                           // Сумматор
10:    SAMSCalculator::aTape                Tape;
                                           // Лента
11:
12:    SAMSCalculator::aController          Calculator
                                           // Калькулятор
13:    (
14:        ExternalInterface,

```

```
15:             Accumulator, // Сумматор
16:             Tape // Лента
17:         );
18:
19:     return Calculator.Operate();
                               // Калькулятор.Работайте()
20: }
```

Анализ

В строках 8–10 создаются экземпляры трех классов, в которых нуждается `aController`.

В строках 12–17 определяется экземпляр `aController`, который называется `Calculator` (Калькулятор), причем ему передаются `ExternalInterface`, сумматор `Accumulator` и лента `Tape`.

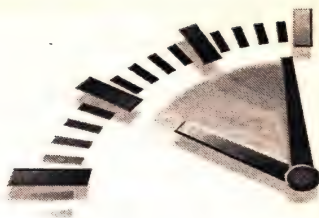
Строка 19 запускает калькулятор `Calculator` и возвращает любой генерируемый им код.

Резюме

Обсуждая реализацию калькулятора, вы научились использовать класс векторов `vector` Стандартной библиотеки C++ и изолировать все контакты с внешним миром в `anExternalInterface`, а также изменили `main.cpp`.

УРОК 21

Перегрузка функций и операторов



В этом уроке вы научитесь давать нескольким разным функциям класса одинаковые названия (имена). Кроме того, вы научитесь давать свою собственную реализацию таким стандартным операторам, как <<.

Объявление перегруженных членов-функций в классе

Идея перегрузки состоит в том, чтобы использовать то же самое название (имя) для нескольких функций класса.

Об этой идее стоит вспомнить тогда, когда несколько функций с разными названиями (именами) выполняют по существу то же самое, а отличаются лишь количеством параметров или их типами. Например, `anExternalInterface` имеет члены-функции `DisplayText()`, `DisplayRequest()` и `DisplayNumber()`. Разве не лучше было бы дать всем этим функциям имя `Display()` — пусть бы компилятор размышлял, изучая параметры, которую из реализаций следует вызвать?

C++ позволяет довольно просто осуществить эту идею.

Для компилятора, например, следующие объявления представляют отдельные и четко отличимые функции-члены класса `anExternalInterface`, несмотря на то, что они имеют то же самое имя:

```
1: void Display(const char *theText) const;  
2: void Display(const aRequest &theRequest) const;  
3: void Display(const float theNumber) const;
```

Определив эти функции, вы можете вызывать их:

```
1: myExternalInterface.Display("Some text");
2: myExternalInterface.Display(theRequest);
3: myExternalInterface.Display(1.5);
```

Эти инструкции безошибочно компилируются и работают, причем ошибок во время выполнения программы не возникает. Управление передается подходящей реализации без каких бы то ни было забот со стороны программиста.

Так что воспользоваться перегрузкой очень просто. (По крайней мере, освоиться с основами перегрузки не сложнее, чем с пультом телевизора.)



Как компилятор находит подходящую функцию?

Компилятор сопоставляет вызов функции-члена с ее реализацией, при этом учитывается класс объекта, название (имя) вызываемой функции, последовательность типов фактических параметров в данном вызове функции (все это называется *сигнатурой вызова* функции). Эта сигнатура вызова сравнивается с сигнатурами реализаций (класс, название (имя) функции, последовательность типов формальных параметров для каждой реализации). Будет вызвана та реализация, чья сигнатура ближе всего к сигнатуре вызова.

Однако если в нескольких функциях последовательности типов параметров очень похожи, т.е. практически одинаковые типы параметров встречаются в том же самом порядке, выбор наилучшего соответствия, сделанный компилятором, может оказаться для вас несколько неожиданным.

Чтобы отличить перегруженные функции, есть несколько альтернативных способов. Применение их гарантирует, что результаты не будут неожиданными. Эти способы представлены ниже.

- Попробуйте сделать так, чтобы типы параметров перегруженных функций отличались настолько, насколько это возможно. Пусть, например, мы имеем следующие объявления функций:

```
1: void SomeFunction
2:     (
3:         const int theFirst,    // константа int
```

```

4:      const int theSecond,    // константа int
5:      const long theThird    // константа long
6:  );

```

И

```

1: void SomeFunction
2:  (
3:      const int theFirst,    // константа int
4:      const long theSecond, // константа long
5:      const int theThird    // константа int
6:  );

```

Эти объявления очень похожи, и в некоторых случаях может быть трудно решить, какую же именно функцию нужно вызвать в вызове вроде

```
1: SomeFunction(3,4,5);
```

В этой ситуации следует пересмотреть перегрузку и просто привести все параметры к “наименьшему общему знаменателю”, т.е. к такому типу, который годится и для int, и для long:

```

1: void SomeFunction
2:  (
3:      const long theFirst,    // константа long
4:      const long theSecond,  // константа long
5:      const long theThird    // константа long
6:  );

```

- Вы можете изменить порядок параметров, чтобы отличать перегруженные функции. Например,

```

1: void SomeFunction
2:  (
3:      const const int theFirst, // константа int
4:      const int theSecond,     // константа int
5:      char theThird            // СИМВОЛ
6:  );
7:
8: void SomeFunction
9:  (
10:     const int theFirst, // константа int
11:     long theSecond,    // константа long
12:     char theThird      // СИМВОЛ
13: );

```

МОЖНО ЗАМЕНИТЬ НА

```

1: void SomeFunction
2:  (
3:      const int theFirst,    // константа int
4:      const int theSecond,  // константа int

```



```
5:      char theThird           // СИМВОЛ
6:    );
7:
8: void SomeFunction
9:    (
10:     char theThird,           // СИМВОЛ
11:     long theSecond,          // КОНСТАНТА long
12:     const int the first // КОНСТАНТА int
13:    );
```

При этом в вызывающей программе вызов второй функции отличается тем, что символ является первым параметром, а не последним.

- Старайтесь сохранять небольшое количество параметров. Одна из целей создания объектов, в отличие от процедурных программ, состоит в том, чтобы усилить возможности объекта так, чтобы он поддерживал внутреннее состояние. Нет ничего неправильного, например, в такой последовательности вызовов:

```
1: SomeObject.SetFirst(3);
2: SomeObject.SetSecond(4);
3: SomeObject.SetThird(5);
4: SomeObject.DoSomething();
```

или в такой:

```
1: SomeObject.SetFirst(3.2);
2: SomeObject.SetSecond("4.2");
3: SomeObject.SetThird(5);
4: SomeObject.DoSomething();
```

Эти вызовы усиливают возможности перегрузки, а также возможности объекта по поддержке внутреннего состояния. Когда вызывается `DoSomething()`, она работает с любыми значениями, которые были установлены предшествующим вызовом функции для данного объекта.

Имейте в виду, что в C++ тип возвращаемого значения функции не рассматривается как часть сигнатуры. Из-за этого следующие два объявления имеют ту же самую сигнатуру, и компилятор сгенерирует сообщение об ошибке:

```
1: void Display(const char *theText) const;
   // КОНСТАНТА-СИМВОЛ
2: int Display(const char *theText) const;
   // КОНСТАНТА-СИМВОЛ
```



Разрешение при перегрузке

Установление соответствия сигнатуры вызова функции с реализацией (определением) функции. Этот процесс также называется *разрешением*. В C++ это разрешение происходит во время компиляции; компилятор генерирует код, который делает соответствующий вызов или генерирует сообщение об ошибках, если вызов не может быть разрешен.

Заданные по умолчанию параметры могут вызывать проблемы при перегрузке. Например, вообразите в `anExternalInterface` две функции со следующими сигнатурами:

```
1: void Display(const char *theText) const;
    // константа-символ
2:
3: void Display
4: (
5:     const char *theText,           // константа-символ
6:     const int thePadWidth = 12     // константа int
7: ) const;
```

Компилятору будет трудно решить, как интерпретировать вызов

```
1: ExternalInterface.Display("Stuff");
```

Ведь его можно интерпретировать так:

```
1: void Display(const char *theText) const;
    // константа-символ
```

или так:

```
1: void Display
2: (
3:     const char *theText,           // константа-символ
4:     const int thePadWidth = 12     // константа int
5: ) const;
```

потому что наличие значения по умолчанию для второго аргумента может означать, что вы намеревались вызвать вторую функцию, но указали только один аргумент. Так что сигнатура вызова соответствует обеим реализациям.

Компилятор позволит скомпилировать эти функции в классе. Но когда программа фактически вызовет эту функцию, причем передаст ей в качестве фактического параметра только строку-литерал, вы получите сообщение компилятора, которое напоминает следующее:

```
[C++ Error] Ambiguity between  
'SAMSCalculator::anExternalInterface::Display  
(const char *) const' and  
'SAMSCalculator::anExternalInterface::Display  
(const char *,const int) const'
```

```
[Ошибка C++] Двусмысленность между  
'SAMSCalculator::anExternalInterface::Display  
(const char *) const' и  
'SAMSCalculator::anExternalInterface::Display  
(const char *,const int) const'
```

Здесь компилятор говорит вам, что он не может увидеть различие между двумя функциями (неоднозначность перегрузки). Ничего нельзя сделать, разве что убрать аргумент по умолчанию.

Вместо аргумента по умолчанию, определите две функции:

```
1: void Display(const char *theText) const;  
2:  
3: void Display  
4:     (  
5:         const char *theText,  
6:         const int thePadWidth  
7:     ) const;
```

Теперь нужно сделать так, чтобы в первой функции "PadWidth" принимала значение 12.

Перегруженные конструкторы

Изучая материал урока 17, "Классы: структуры с функциями", вы уже встречались с перегрузкой конструктора и конструктора копии для `aRequest`.

Конструктор — член-функция с названием (именем), которое совпадает с названием (именем) класса. Подобно любой перегруженной член-функции, она имеет сигнатуру, которую компилятор использует для разрешения перегрузки. Как и другие перегруженные функции, перегруженные конструкторы не должны иметь аргументов по умолчанию.

Что означает перегрузить оператор?

В C++, как вы уже знаете, предусмотрен богатый и разнообразный набор операторов — от таких простых, как `+` и `-`, до таких, как `+=`, `<<` и `++`. Большинство этих операторов имеют четкий смысл и применимы они к довольно широкому диапазону типов, но ни один из них не применим к определенным пользователем типам вроде структур и классов. Это и не удивительно. В конце концов, что может означать сложение двух экземпляров класса `aRequest`?

С другой стороны, как вы уже видели в примере, в котором использовались строки из стандартного пространства имен (строки `std::string`), чтобы собрать сообщение для `runtime_error` (см. урок 11, “Переключатели (инструкции выбора `switch`), статические переменные и ошибки во время выполнения”), может быть очень удобно “сложить” два объекта (экземпляра класса). В том примере результатом был строковый (типа `string`) объект, который являлся суммой (строго говоря, конкатенацией) двух строковых (типа `string`) объектов — строкового объекта с правой и строкового объекта с левой стороны оператора `+`.

Вы также видели, что объект `cout` перегружает оператор вставки `<<` (иногда называемый *оператором сдвига влево*). Этот перегруженный оператор позволяет писать инструкции вроде

```
1: cout << "This" << Number << " is OK." << endl;
```

Наконец, иногда полезно перегрузить операторы отношения (сравнения) так, чтобы экземпляры класса можно было сравнивать и сортировать.

Этот стиль кодирования можно считать естественным для некоторых классов. Если вы хотите использовать его, вы должны перегрузить операторы. Почти каждый оператор C++ может быть перегружен в классах, созданных пользователем. Нужно только знать, как это сделать.

Перегрузка оператора может быть опасна

Перегрузкой операторов очень часто злоупотребляют. Кроме того, ее не менее часто неправильно используют в C++. Программист несет ответственность за неправильное использование символа оператора, т.е. за такое использование, которое противоречит интуитивному пониманию значения оператора другим программистом. Например, не перегружайте +, чтобы он означал вычитание, и реализуйте ! так, чтобы он делал число отрицательным. Кроме того, вы должны понятно и четко объяснить, что делает любой перегруженный оператор, написав комментарий в заголовочном файле там, где оператор объявлен.

Вообще говоря, код будет намного более удобочитаемым, если в нем использовать обычные член-функции с осмысленными названиями (именами), а не перегруженные операторы.

Перегрузка оператора <<

Если бы в модуле `anExternalInterface` был предусмотрен оператор <<, то использовать этот модуль было бы несколько легче, поскольку тогда можно было бы использовать `anExternalInterface` так, как вы используете `cout`. Например, можно было бы написать:

```
myExternalInterface << "Presents: " << myTape.Element(Index);
```

Добавить эту возможность довольно легко. Начните с объявления оператора << для `const char *` (т.е. строк-литералов в стиле C++) в качестве члена `anExternalInterface`, как показано в листинге 21.1.

Листинг 21.1. Перегрузка оператора << в `anExternalInterface`

```
anExternalInterface &operator << (const char *theText);
```

Имейте в виду, что это совсем не нечто волшебное; вы просто объявляете функцию с несколько странным названием (именем) (`operator <<` — оператор <<).

Анализ

Первая часть этой строки — тип значения, возвращаемого оператором (т.е. членом-функцией). Обычно вы объявляете, что оператор (член-функция) возвращает ссылку на объект класса, членом которого он является. Фактически функция-оператор должна вернуть ссылку на экземпляр класса, для которого она была вызвана.

Это возвращаемое значение позволяет применить следующий оператор в последовательности операторов к тому же самому объекту, к которому применялся первый.

Если функцию `Display()` модуля `anExternalInterface` объявить так:

```
1: anExternalInterface &Display(const char *theText)
                                     const;
2: anExternalInterface &Display
3:   (const aRequest &theRequest) const;
4: anExternalInterface &Display(const float theNumber)
                                     const;
```

причем если бы каждая функция возвращала экземпляр, для которого она вызывалась, то можно было бы написать:

```
1: myExternalInterface.Display("Presents: ").
1:⚡ Display(myTape.Element(Index));
```

Этот код мог бы вывести следующий результат:

Presents: +34

Поскольку оператор-функция `<<` возвращает ссылку на ее экземпляр, следующий фрагмент делал бы то же самое:

```
1: myExternalInterface << "Presents: " <<
                                     myTape.Element(Index);
```

Следующий элемент в объявлении функции-члена, являющейся оператором, — специальное ключевое слово `operator` (оператор), которое сообщает компилятору, что название (имя) функции — не обычное слово, и что компилятор должен искать следующий символ в его таблице операторов. Это заставит компилятор удостовериться, что символ указан правильно, и определить, сколько параметров он должен иметь. В нашем случае имеется один параметр — текст.

Заключительная часть объявления функции-члена, являющейся оператором, — параметр функции-оператора. В нашем случае это `const char *`, но это могло быть чем угодно.



Функции-члены, которые являются операторами, вызываются иначе

Для функций-членов, которые являются операторами, не нужно использовать операторы выбора члена, чтобы указать, что они являются членами класса, к которому относятся их левый операнд, и при этом им не нужны круглые скобки вокруг их операнда. Компилятор распознает их как вызовы функций без таких художественных оформлений. Однако иногда проще понять вызовы функции-оператора, если представить их так, как будто они являются обычными вызовами, например:

```
myExternalInterface.<<(* Presents: *)>>
(myTape, Element (Index));
```

Это не будет компилироваться, но оно передает смысл более привычным способом.



Функции-члены, которые являются операторами, могут иметь не больше одного параметра

Функции-члены, которые являются операторами, перегружают или одноместные (унарные), или инфиксные операторы. Если перегружаемый оператор одноместный (например ! или ++), он не имеет никаких параметров и применяется непосредственно к экземпляру класса и никакая дополнительная информация не нужна. Если оператор инфиксный (+, +=, << или !=), он имеет один параметр, который должен иметь тип объекта, ожидаемого в выражении справа от оператора.

Поскольку член-функцию, которая является инфиксным оператором, можно перегрузить, то для любого оператора могут быть объявлены несколько операторных функций-членов, причем все они должны отличаться типом параметра.

По умолчанию перегрузка одноместного оператора, который может быть префиксным или постфиксным, происходит для префиксной версии. Чтобы перегрузить постфиксную версию, необходимо указать фиктивный параметр. Например, чтобы перегрузить постфиксный оператор приращения, напишите следующее:

```
aSomeClass &operator++(int theIgnored);
```

Итак, мы закончили объявление перегруженного оператора <<. Сейчас мы должны определить его в файле реализации. То, как это делается, продемонстрировано в листинге 21.2.

Листинг 21.2. Реализация оператора <<
в `anExternalInterface`

```

1: anExternalInterface &anExternalInterface::operator <<
1:  (const char *theText) const
2: {
3:     Display(theText);
4:     return *this;
5: };

```

Анализ

Строка 1 — заголовок реализации функции, который идентичен прототипу функции, за исключением того, что он содержит `имя_класса::` перед именем функции (которое представляет собой `operator <<` — оператор <<). Как обычно, конструкция `имя_класса::` указывает, что функция — член класса. В нашем примере этим классом является `anExternalInterface`.

Чтобы отобразить текст, в строке 3 записано обращение к перегруженной функции `Display()`.

В строке 4 возвращается ссылка на текущий экземпляр `anExternalInterface`. Специальное ключевое слово `this` (это) является указателем на экземпляр объекта, для которого эта функция вызывалась. Символ `*` имеет свой обычный смысл — разыменование указателя, чтобы результат можно было присвоить ссылке. По существу, строка 4 эквивалентна `anExternalInterface &ThisInstance = *this;`
`return ThisInstance;`

Теперь вы перегрузили один оператор. Точно так же вы можете создать дополнительные перегруженные операторы вставки для других функций `Display()`:

```

anExternalInterface &operator << (const aRequest
&theRequest);
anExternalInterface &operator << (const float theNumber);

```

Чтобы перегрузить другие операторы, вы можете следовать приведенному выше образцу.

Ключевое слово `const` (константа) и перегруженные операторы

Большинство член-функций было объявлено с ключевым словом `const` (константа), но это не относится к операторной член-функции, которую вы только что определили для `anExternalInterface`. Почему ее объявление отличается от объявлений большинства член-функций?

Обычно операторы создаются с намерением изменить состояние объекта, для которого они вызываются. Но в случае с `anExternalInterface`, оператор вставки не изменяет его состояние. Поэтому мы можем и должны “сделать его константой”, т.е. применить к нему ключевое слово `const` (константа).

Это довольно просто сделать в прототипе функции — как обычно, нужно только прибавить ключевое слово `const` (константа) в конце объявления.

В файле реализации прибавьте ключевое слово `const` (константа) в конце заголовка функции.

В определении функции есть только одна небольшая хитрость — она отмечена в листинге 21.3.

Листинг 21.3. Как в `anExternalInterface` сделать оператор << константой

```
1: anExternalInterface &anExternalInterface::operator <<
1:  (const char *theText) const
2: {
3:     Display(theText);
*4:     return const_cast<anExternalInterface &>(*this);
5: };
```

Анализ

В строке 4 для результата `*this` выполняется обычная операция, называемая `const_cast`. Благодаря этой операции результат `*this` (разыменованный указатель никогда не является константой (`const`)), который не является константой (`const`), превращается в ссылку-константу (`const`) на `anExternalInterface` (как определено в угловых скобках).

Влияет ли это как-нибудь на программу или экземпляр объекта? Нет, цель этого преобразования состоит в том, чтобы “успокоить” компилятор. Это означает: “Да, я как про-

граммист знаю то, что я делаю, и я подтверждаю, что я не изменил этот объект любым обнаруживаемым способом, и я не хочу, чтобы программа, вызывающая операторную функцию-член, “подумала”, что я внес какие-либо изменения”.

Перегрузка: ключевые положения

Запомните следующие ключевые положения.

- Перегружать можно любую функцию, которая имеет параметры.
- Просто объявите, определите и используйте столько перегрузок функции, сколько вам нужно. Перегрузки отличаются типами аргументов, их количеством и порядком. Но будьте осторожны — избегайте чрезмерного количества аргументов, попытайтесь сделать так, чтобы типы аргументов отличались как можно больше, и, если все другие ухищрения все еще не привели к четким отличиям в типе аргументов, измените порядок типов аргументов в перегруженных функциях.
- Возвращаемый функцией тип не изменяет сигнатуру функции и не может использоваться для того, чтобы различить перегруженные функции при разрешении.
- Перегружать можно любой оператор — и одноместный (префиксный или постфиксный), и инфиксный.
- Одноместные (унарные) операторы могут иметь не более двух перегрузок для каждого класса — одну для префиксного и одну для постфиксного.
- Инфиксные операторы могут иметь столько перегрузок, сколько есть используемых для перегрузки типов параметров, но не больше, потому что они имеют только один параметр.
- Везде, где возможно, избегайте перегрузки операторов. Вместо них используйте функции с читаемыми названиями (именами).

Перегрузка присваиваний и конструктора копии

Есть одна область, в которой ничто не может справиться с работой лучше, чем перегрузка оператора. Вы уже знаете, как используется конструктор копии. Обычно он необходим только тогда, когда поля указывают на динамическую память. И вызывается он также только тогда, когда объект создается (явно или неявно) как часть копии (например, когда локальная переменная возвращается из функции-члена).

Присваивание — это совсем не то же самое, что создание копии, поскольку присваивание может происходить в любой момент во время существования объекта, а не только в момент его создания. Например:

```
aRequest Request1(aRequest::add, 34);  
aRequest Request2(aRequest::multiply, 22);  
Request2 = Request1;
```

После выполнения последней инструкции вы ожидаете, что Request2 будет иметь оператор (извлекаемый функцией `Operator()`) `aRequest::add` (добавить) и операнд (извлекаемый функцией `Operand()`) 34.

Таким образом, нужно создать не только конструктор копии, но и перегрузить оператор присваивания. Это особенно нужно в том случае, если некоторые из ваших полей используют динамическую память (кучу). Как и конструктор копии по умолчанию, присваивание по умолчанию делает почленную копию. Таким образом, если члены хранят адреса динамической памяти, присваивание, подобно копированию, приведет к наличию двух объектов, совместно использующих те же самые адреса членов-данных (полей), хранящихся в динамической памяти.

Важно избегать еще одной проблемы, которая может возникнуть при присваивании: присваивание объекта самому себе.

```
aRequest Request2(aRequest::multiply, 22);  
Request2 = Request2;
```

Хотя это походит на глупую ошибку, но при наличии ссылок и указателей ее очень просто допустить. Поэтому главное правило при перегрузке операторов присваивания гласит: всегда нужно удостовериться, что вы не присваиваете объект

самому себе. Недостаточная осторожность может привести к различным трудным проблемам.

В листинге 21.4 показан пример функции-члена, которая является оператором присваивания для `aRequest`, с условным оператором `if`, предотвращающим эту “глупую ошибку”.

Листинг 21.4. Перегрузки оператора `=` в `aRequest`

```

1: aRequest &operator = (aRequest &theOtherRequest)
2: { // Если (это != &theOtherRequest)
3:     if (this != &theOtherRequest)
4:     {
5:         myOperator = theOtherRequest .myOperator;
6:         myOperand = theOtherRequest .myOperator;
7:     };
8:
9:     return *this;
10: }
```

Анализ

Строка 3 выясняет, расположен ли `theOtherRequest` по тому же адресу, на который указывает `this` (это). Если это так, значит, `this` (это) — тот же самый объект, что и `theOtherRequest`, и присваивание не выполняется.

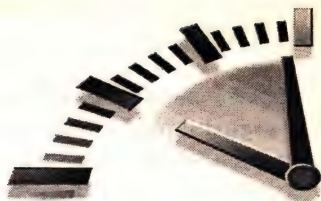
Точно таким же образом рекомендуется использовать условный оператор `if` в конструкторе копии.

Резюме

Вы научились создавать несколько функций с тем же самым названием (именем) и разными типами параметров. Вы также узнали, как компилятор поддерживает перегрузку названий (имен) таких функций. Кроме того, вы научились перегружать операторы; в частности, перегрузили операторы вставки и присваивания, но, несомненно, при необходимости вы сможете перегрузить и многие другие.

Вы получили предупреждения о некоторых из ловушек перегрузки и, надеемся, будете внимательны, когда вам придется использовать перегрузку.

УРОК 22



Наследование

В этом уроке вы познакомитесь с наследованием — средством создания классов на основе других классов в целях добавления в них новых членов и изменения реализации, благодаря чему созданные классы могут выполнять более специализированные задачи.

Объявление наследования

Вплоть до этого момента вы использовали средства языка C++, ориентированные на работу с объектами. Однако полноценный объектно-ориентированный язык должен поддерживать еще и *наследование*.

Наследование — средство создания новых классов, чьи возможности заимствованы не менее чем из одного другого класса (часто называемого *суперклассом* или *предком*). Наследование называется также “программированием отличий”, потому что в новом классе (называемом *производным классом* или *потомком*) нужно запрограммировать только тот код, который отличает производный класс от суперкласса.

Наследование позволяет заменить или прибавить функции-члены или прибавить поля к существующему классу путем создания производного класса, в который как раз и помещаются изменения (или замены). В большинстве случаев все это можно сделать без изменения суперкласса.

В этом уроке вы примените наследование для усовершенствования классов калькулятора. Усовершенствования восстановят способность записывать ленту в поток в конце выполнения программы и читать ее в начале выполнения.

Новые и измененные классы

UML-диаграмма калькулятора, показанного на рис. 22.1 (новая версия рис. 18.1), содержит новые и измененные классы, выделенные затененными полями. Два новых класса — `aPersistentTape` и `aPersistentTapeExternalInterface`. “Persistent” означает “постоянный”.

В качестве суперкласса для `aPersistentTapeExternalInterface` был взят `anExternalInterface`. Кроме того, чтобы сделать использование некоторых возможностей классами `aPersistentTape` и `aPersistentTapeExternalInterface` более безопасным, эти возможности были перемещены из `anExternalInterface` в `aRequest`. Изменения также включали перемещение функций, которые в `aRequest` транслировали `aRequest::anOperator` в символ и обратно. Эти изменения будут обсуждаться в уроке 24, “Абстрактные классы, множественное наследование и статические члены”.



Изображение наследования в UML

В UML стрелки идут от производного класса к его суперклассу; поэтому стрелку можно читать как “наследует от”.

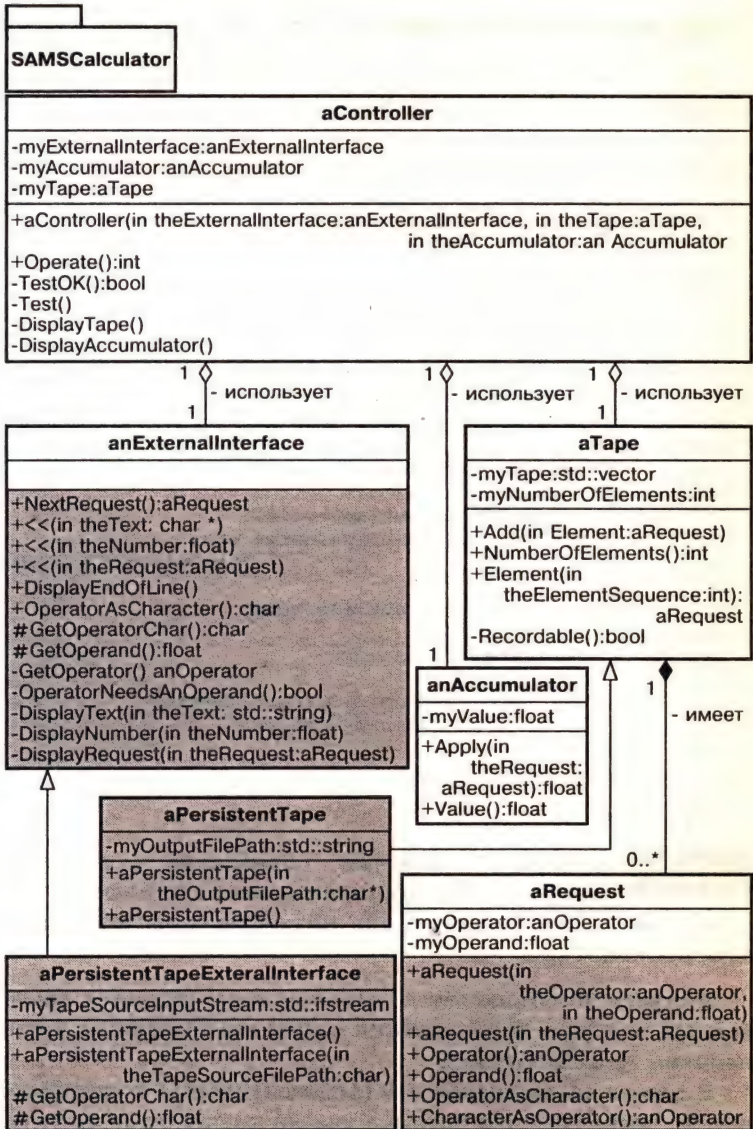


Рис. 22.1. UML-диаграмма класса с наследованием

Создание производного класса ленты Tape

Наследование объявить просто. Листинг 22.1 — новое объявление для `aPersistentTape`.

Листинг 22.1. Заголовок `aPersistentTape`

```

1: #ifndef PersistentTapeModuleH
2: #define PersistentTapeModuleH
3: // строки
4: #include <string>
5:
6: #include "TapeModule.h"
7:
8: namespace SAMSCalculator // пространство имен
9: { // класс aPersistentTape: общедоступный aTape
10:     class aPersistentTape: public aTape
11:     {
12:     public: // общедоступный
13:
14:         aPersistentTape(void);
15:         aPersistentTape(const char
                                *theOutputFilePath);
16:
17:         ~aPersistentTape(void);
18:
19:     private: // частный
20:
21:         std::string myOutputFilePath;
                // станд.: строка
22:     };
23: };
24:
25: #endif

```

Анализ

Строка 6 включает объявление суперкласса `aTape`, чтобы компилятор имел всю информацию о его членах, необходимую для интерпретации объявления `aPersistentTape`.

В строке 10 объявляется, что этот класс является производным от `aTape`. В объявлении класса это обозначено добавлением: `public aTape`.

В строке 14 *переопределен* (заменен) старый конструктор `aTape`, поскольку для потомка указан конструктор по умолчанию.

В строке 15 *перегружается* (фактически добавляется) конструктор, которого не было в `aTape`. Этот новый конструктор сохраняет `theOutputFilePath`, который будет использоваться как адресат при записи ленты в файл; путь будет сохранен в `myOutputFilePath`.

В строке 17 к классу добавляется деструктор. Лента будет записана в файл перед разрушением экземпляра, т.е. тогда, когда вызывается эта функция.

В строке 21 добавляется новое поле `myOutputFilePath`.

Все конструкторы, функции и поля суперкласса являются частью производного класса. Но ни один из этих элементов даже не упоминается в потомке, если он не изменяется.

Реализация производного класса

Реализация приведена в листинге 22.2. В реализации есть только тот код, который добавляет возможности к коду в суперклассе.

Листинг 22.2. Реализация `aPersistentTape`

```

1: #include <fstream>
2: #include <exception>
3: #include <string>
4:
*5: #include "PersistentTapeModule.h"
6:
7: namespace SAMSCalculator // пространство имен
8: { // использовать станд. пространство имен
9:     using namespace std;
10:
*11:     aPersistentTape::aPersistentTape(void)
*12:     {
*13:         throw
*14:             runtime_error
*15:             ( // строка - конструктор
               // для aPersistentTape
*16:               string("The default constructor for
*16: aPersistentTape has been used. ") +
*17:               string("Use only the constructor that
*17: requires the file path.")
*18:             ); // обязательно требуется конструктор
               // с файлом
*19:     };
20:

```



```

21:      aPersistentTape::aPersistentTape
21:  ↗      (const char *theOutputFilePath):
*22:      myOutputFilePath(theOutputFilePath)
23:      {
24:      };
25:
*26:      aPersistentTape::~aPersistentTape(void)
*27:      {
*28:          if (myOutputFilePath.size() > 0)
*29:          {
*30:              ofstream OutputStream
*30:  ↗              (myOutputFilePath.c_str(), ofstream::out);
*31:
*32:              int NumberOfTapeElements =
                  NumberOfElements();
*33:
*34:              for
*34:  ↗              (
*34:  ↗                  // цикл
*34:  ↗                  int Index = 0; // Индекс = 0
*34:  ↗                  Index < NumberOfTapeElements;
*34:  ↗                  Index++ // Индекс: ++
*34:  ↗              )
*35:              {
*36:                  OutputStream << // Элемент (Индекс)
*36:  ↗                  Element(Index).OperatorCharacter() <<
*36:  ↗                  Element(Index).Operand();
*37:              };
*38:          };
*39:      };
40: };

```

Анализ

Как обычно, для этого класса включается заголовочный файл — это делает строка 5.

Строки 11–19 предназначены для того, чтобы предотвратить использование конструктора по умолчанию. Если вместо нового конструктора используется конструктор по умолчанию, будет вызвано исключение. Это необходимо, потому что суперкласс имеет конструктор по умолчанию, а наследование не позволяет производным классам устранять что-либо унаследованное от их предков. Так что нельзя избавиться от конструктора по умолчанию `aTape`; его нужно скрыть путем отмены.

Строка 22 инициализирует значением `theOutputFilePath` переменную `myOutputFilePath` производного класса, в которой хранится название (имя) файла для вывода.

Строки 26–39 — деструктор `aPersistentTape`. Он будет вызываться при выходе экземпляра из области видимости

или, если экземпляр был создан с помощью операции `new` (новый, создать), при его разрушении с помощью операции `delete` (удалить). В строке 32 используется член-функция `NumberOfElements()` суперкласса для получения размера ленты. В строке 36 для записи конкретного элемента в поток используется член-функция `Element()`, также принадлежащая суперклассу.

Эти вызовы член-функций суперкласса не требуют никаких специальных ключевых слов или обозначений. Члены суперкласса являются составной частью потомка и могут использоваться точно так же, как любые члены, объявленные в потомке. Компилятор знает, откуда нужно брать функции и найдет подходящую реализацию.



Будьте осторожны при переопределении члена суперкласса

Если в потомке переопределена функция суперкласса, то вызов в потомке по умолчанию вызовет переопределенную функцию, а не функцию суперкласса. Это может привести к появлению трудноуловимых ошибок. Будьте осторожны.

В нашей программе есть одно ограничение на наследование: нужно использовать именно эти функции, а не член-переменную `myTape`, потому что `myTape` является частной (`private`) переменной `aTape`, а частные (`private`) члены суперкласса скрыты от любого другого класса, даже от производного. Такое сокрытие очень полезно, потому что благодаря ему при изменении суперкласса изменения в производном классе не понадобятся.

Ссылки на объект своего класса и суперкласса

В листинге 22.3 приведен новый код `main.cpp`. Он свидетельствует о том, что в результате наследования ссылка на суперкласс может использоваться как ссылка на класс-потомок.

Листинг 22.3. main.cpp. Использование новых классов

```

*1: #include "PersistentTapeExternalInterfaceModule.h"
*2: #include "AccumulatorModule.h"
*3: #include "PersistentTapeModule.h"
*4: #include "ControllerModule.h"
*5:
*6: int main(int argc, char* argv[]) // главная программа
*7: {
*8:     SAMSCalculator::aPersistentTapeExternalInterface
*9:     ExternalInterface(argv[1]);
*10:    SAMSCalculator::anAccumulator
*11:    Accumulator; // Сумматор
*12:    SAMSCalculator::aPersistentTape Tape(argv[1]);
*13:                                     // Лента
*14:
*15:    SAMSCalculator::aController
*16:    Calculator // Калькулятор
*17:    (
*18:        ExternalInterface,
*19:        Accumulator, // Сумматор
*20:        Tape // Лента
*21:    );
*22:
*23:    return Calculator.Operate();
*24:                                     // Калькулятор.Работайте
*25: }

```

Анализ

Строки 1 и 3 включают новые модули для производных классов.

Строки 8 и 10 теперь определяют экземпляры `aPersistentTape` и `aPersistentTapeExternalInterface` на месте старых переменных, в которых хранились экземпляры классов `aTape` и `anExternalInterface`. Эти переменные используют новые специальные конструкторы производных классов.

Если взглянуть на рис. 22.1, то будет понятно, что `aController` не был изменен. Его конструктор в качестве параметров по-прежнему принимает ссылки на¹ `aTape` и `anExternalInterface`, причем он сохраняет эти ссылки в качестве полей, которые используются для вызова `Operate()`.

Это демонстрирует специфическое свойство наследования, называемое *полиморфизмом класса*. Полиморфизм класса —

¹ Экземпляры классов, конечно, в соответствии с соглашением об именовании. — *Прим. ред.*

это свойство, которое состоит в том, что ссылка на класс-потомок может использоваться там же, где и ссылка на суперкласс. Например, объект `aPersistentTape` можно передать через параметр-ссылку на `aTape` и его можно присвоить переменной-ссылке на `aTape`. В дальнейшем вы увидите, что при этом объект не теряет возможности, добавленные потомком. Однако это означает, что не придется изменять те части программы, которые не используют новых общедоступных членов производного класса. Это значительно уменьшает риск добавления ошибок при сопровождении.



Полиморфизм, указатели и ссылки

Полиморфизм класса зависит от использования указателей и ссылок. Это одна из причин, по которой так важно избегать передачи объектов фактического класса в качестве параметров, а также избегать сохранения копии экземпляра. Используйте ссылки везде, где это возможно.



Полиморфизм

В C++ полиморфизмом называется способность иметь несколько реализаций с тем же самым названием (именем). C++ поддерживает три типа полиморфизма: полиморфизм методов (перегруженные функции), полиморфизм операторов (перегруженные операторы) и полиморфизм классов, который состоит в том, что можно сослаться на класс-потомок так же, как и на его суперкласс.

Полиморфизм класса работает потому, что производный класс может делать все то, что может делать суперкласс и даже более того.

Уровни наследования

Производный класс может быть получен в результате не одного, а нескольких наследований. Тогда говорят, что он находится не на первом, а на более высоком уровне наследования. В C++ количество уровней наследования не ограничено, и нет ничего необычного в том, что в полноценной библиотеке классов некоторые производные классы имеют четыре или пять уровней наследования (считая от суперкласса). Кроме

того, от того же самого суперкласса могут наследовать несколько классов. По этой причине программисты часто рассматривают иерархию наследования как *дерево наследований*. Это — инвертированное (перевернутое) дерево с корнем наверху, как показано на рис. 22.2.

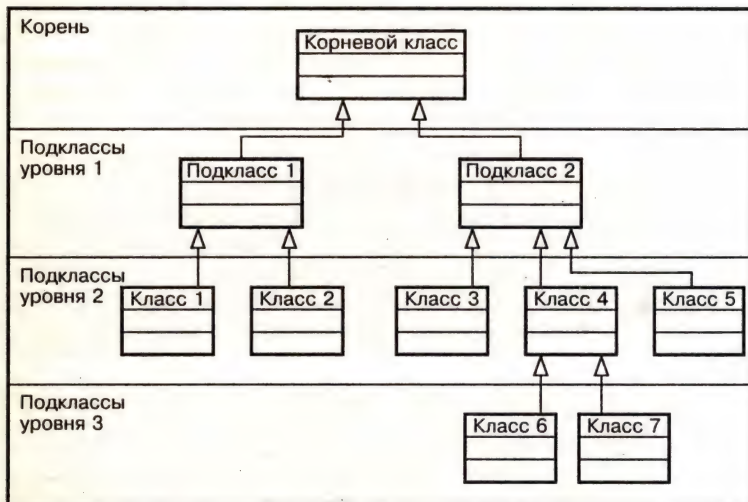


Рис. 22.2. Дерево наследований

Переопределение функций

Как вы знаете, в производном классе можно переопределить функции его суперкласса. Функции-члены производного класса могут даже вызывать функции суперкласса — включая и те реализации, которые они переопределяют. Возможные варианты показаны на рис. 22.3.

На этой диаграмме показан объект (экземпляр класса), который состоит из производного класса и его суперкласса. Когда функция вне объекта обращается к `SomeFunction()`, вызов полностью обслуживается производным классом и управление не передается соответствующей реализации суперкласса. Когда функция вне объекта обращается к `OtherFunction()`, из-за того, что производный класс не реализует эту функцию, вызов обслуживается реализацией суперкласса. Когда вызывается

ThirdFunction(), она делегирует вызов суперклассу и, возможно, дополнительно использует часть своего собственного кода, чтобы расширить услуги, предоставляемые функцией ThirdFunction() суперкласса. Наконец, функция NewFunction() является новой в производном классе. Она использует поля (общедоступные (public) поля, конечно) суперкласса и, кроме того, обращается к ThirdFunction() суперкласса (хотя было бы безопаснее обращаться к реализации функции ThirdFunction() в производном классе).

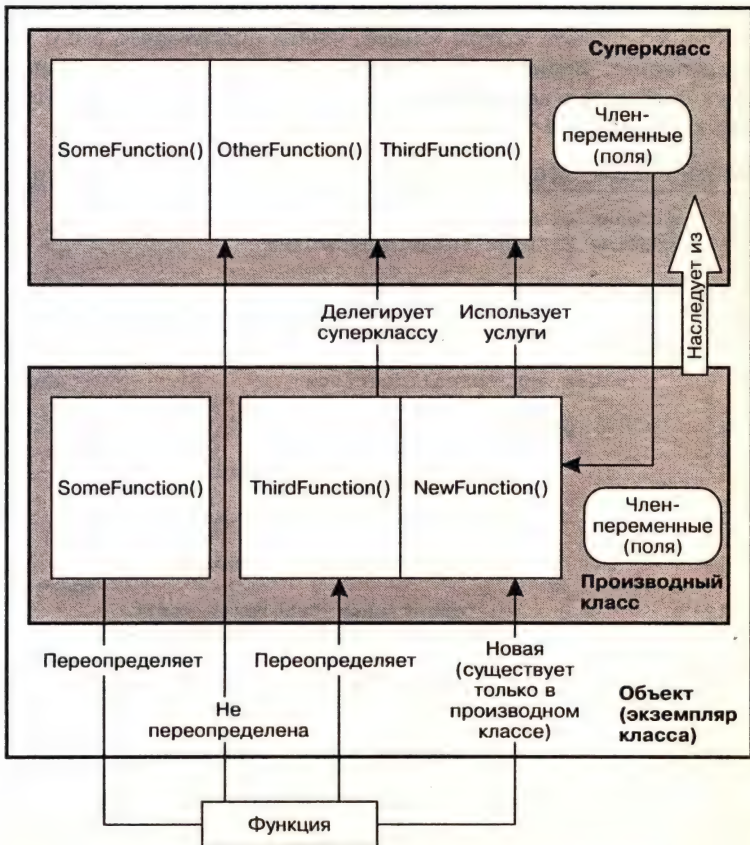


Рис. 22.3. Вызовы функций в производном классе

Защищенный доступ

Как известно, производный класс не может использовать поля и функции, объявленные в частном (private) разделе суперкласса. Но суперкласс может иметь функции и поля, которые могут использовать только суперкласс и его производные классы. Для этого такие функции и поля помещаются в специальном защищенном (protected) разделе.

В объявлении `anExternalInterface`, приведенном в листинге 22.4, показаны некоторые из изменений, которые необходимо сделать в этом классе, чтобы подготовить его к наследованию. В число этих изменений входит и размещение двух функций в защищенном разделе, чтобы их можно было переопределить в `aPersistentTapeExternalInterface`.

Листинг 22.4. Защищенный раздел в `anExternalInterface`

```

1: #ifndef ExternalInterfaceModuleH
2: #define ExternalInterfaceModuleH
3:
4: #include "RequestModule.h"
5:
6: namespace SAMSCalculator           // пространство имен
7: {
8:     class anExternalInterface        // класс
9:     {
10:     public:                          // общедоступный
11:
12:         anExternalInterface(void);
13:
14:         aRequest NextRequest(void) const;
15:
16:         anExternalInterface &operator <<
17:             (const char *theText) const;           // оператор
18:         anExternalInterface &operator <<
19:             (const float theNumber) const;         // оператор
20:         anExternalInterface &operator <<
21:             (const aRequest &theRequest) const;    // оператор
22:
23:         void DisplayEndOfLine(void) const;
24:
25:         char OperatorAsCharacter
26:             (aRequest::anOperator theOperator)
27:             const;

```

```

23:
*24:         protected:                                // защищенный
*25:
*26:         virtual char GetOperatorChar(void);
                                           // виртуальный
*27:         virtual float GetOperand(void);
                                           // виртуальный
*28:
29:         private:                                     // частный
30:
31:         bool OperatorNeedsAnOperand
31:↳         (aRequest::anOperator theOperator)
           const;
32:         aRequest::anOperator GetOperator(void)
           const;
33:
34:         void Display(const char *theText) const;
35:         void Display(const aRequest &theRequest)
           const;
36:         void Display(const float theNumber)
           const;
37:     };
38: };
39:
40: #endif

```

Анализ

Защищенный раздел показан в строках 24–28.

В этих строках определены основные функции для получения символа оператора и операнда с пульта. Эти функции будут переопределены в `aPersistentTapeExternaInterface`.

Что означает ключевое слово **virtual** (виртуальный)?

В строках 26 и 27 листинга 22.4 показаны две функции, квалифицированные с ключевым словом `virtual` (виртуальный). Это важно, если функции нужно переопределить, потому что это — единственный способ удостовериться, что полиморфизм класса работает должным образом.

Когда функция вне объекта вызывает функцию объекта, компилятор генерирует код, который выполняет вызов функции. Если функция не имеет ключевого слова `virtual` (виртуальный), компилятор предполагает, что вызывающая программа хочет вызвать функцию, реализованную в том классе, с которым имеет дело вызывающая программа. К со-

жалению, если вызывающая программа имеет ссылку на суперкласс того класса, который фактически использовался для создания экземпляра объекта, а в производном классе вызываемая функция переопределена, вызывающая программа не будет использовать улучшенную версию функции, т.е. не будет использовать функцию производного класса.

Ключевое слово `virtual` (виртуальный) сообщает компилятору, что функция будет переопределена в производных классах. Компилятор готовится к этому, выполняя любые вызовы этой функции с помощью указателя на функцию. Он следит за значением этого указателя на функцию и делает так, чтобы он всегда указывал на реализацию данной функции в самом глубоком производном классе, в котором эта функция переопределена.

Это особенно мощное средство, потому что оно работает даже для обращений из функций в суперклассе к другим член-функциям суперкласса. Иными словами, функция суперкласса в конце концов вызовет реализацию функции из производного класса, если эта функция имеет ключевое слово `virtual` (виртуальный) и если объект фактически является экземпляром производного класса. Именно это используется в реализации `aPersistentTapeExternalInterface`.

В листинге 22.5 показано использование этой особенности в одном из разделов реализации суперкласса (`anExternalInterface`).

Листинг 22.5. `GetOperatorChar()` в суперклассе `anExternalInterface`

```
*1: char anExternalInterface::GetOperatorChar(void)
*2: {
*3:     char OperatorChar;
*4:     cin >> OperatorChar;
*5:     return OperatorChar;
*6: };
7:
8: aRequest::anOperator
8: ↪ anExternalInterface::GetOperator(void) const
9: {
*10:     return aRequest::CharacterAsOperator
*10: ↪ (GetOperatorChar());
11: };
```


Анализ

Строки 1–6 — реализация `GetOperatorChar()` в суперклассе. В строке 10 вызывается `GetOperatorChar()`. Но в листинге 22.6 функция `GetOperatorChar()` из `aPersistentTapeExternalInterface` переопределена таким образом, что сначала читает ленту с файла.

Листинг 22.6. `GetOperatorChar()` в производном классе `aPersistentTapeExternalInterface`

```

1: char aPersistentTapeExternalInterface::GetOperatorChar
2: (void)
3: {
4:     if                                     // если
5:     (
6:         myTapeSourceInputStream.is_open() &&
7:         !myTapeSourceInputStream.eof()
8:     )
9:     {
10:         char OperatorChar;
11:         myTapeSourceInputStream >> OperatorChar;
12:
13:         if (OperatorChar == '\0')    // если файл пуст
14:         {
15:             myTapeSourceInputStream.close();
*16:             return anExternalInterface::
                GetOperatorChar();
17:         }
18:         else
19:         {
20:             return OperatorChar;
21:         };
22:     }
23:     else
24:     {
25:         if (myTapeSourceInputStream.is_open())
                                                // если открыт
26:         {
27:             myTapeSourceInputStream.close();
28:         };
29:
*30:         return anExternalInterface::GetOperatorChar();
31:     };
32: };

```

Анализ

Когда в новой реализации главной программы `main.cpp` главная функция `main()` передает калькулятору `Calculator` экземпляр класса `aPersistentTapeExternalInterface` и калькулятор вызывает `myExternalIn-`

`terface.NextRequest()`, `NextRequest()` вызывает `anExternalInterface::GetOperator()`, которая в результате виртуальности (ключевое слово `virtual`) вызывает `aPersistentTapeExternalInterface::GetOperatorChar()`, а не `anExternalInterface::GetOperatorChar()`. Это позволяет записать только очень маленькое количество очень специфического кода в `aPersistentTapeExternalInterface` и многократно использовать большую часть кода из `anExternalInterface`. Фактически, в строках 16 и 30 листинга 22.6 `aPersistentTapeExternalInterface` даже делегирует все запросы на ввод с пульта реализации функции в суперклассе.

Виртуальные конструкторы и деструкторы

Ключевое слово `virtual` (виртуальный) никогда не используется для конструкторов, но всегда очень полезно для деструкторов. В противном случае при разрушении производного класса с помощью указателя на суперкласс или ссылки (другими словами, с помощью `delete` (удалить)), может вызываться деструктор суперкласса, и тогда любая память, размещенная в динамической памяти (или другие ресурсы, затребованные производным классом), не будет освобождена должным образом. Тот факт, что деструктор `aTape` является виртуальным, гарантирует, что потомки класса, к которому принадлежит этот деструктор, т.е. потомки класса `aTape`, будут уничтожены безопасным образом. Поскольку “виртуальность” наследуется, `aPersistentTape` имеет виртуальный деструктор с подачи `aTape`.

Виртуальные функции-члены

Ключевое слово `virtual` (виртуальный) должно всегда использоваться для функции в защищенном (`protected`) разделе. Если вы собираетесь переопределить общедоступную (`public`) функцию, она также должна иметь ключевое слово `virtual` (виртуальный).

Некоторые программисты обеспокоены снижением эффективности из-за косвенного вызова виртуальной функции. По правде говоря, вероятно, такое снижение эффективности несущественно для обычных программ в сравнении с такими операциями, как ввод-вывод файлов или доступ к базе данных, и даже в сравнении со временем, которое требуется для печати символа.

Вызов суперкласса

В строке 30 листинга 22.6 содержится обращение к функции `GetOperatorChar()` из `anExternalInterface`. Оно выполняется тогда, когда будут исчерпаны команды в файле ленты. Вызывается функция `GetOperatorChar()` именно из `anExternalInterface` потому, что в вызове функции сначала указано название (имя) класса, в котором функция определена, а за этим именем следует оператор разрешения области видимости.



Ссылка на реализацию суперкласса опасна

Одна из наиболее общих проблем в объектно-ориентированных системах с развивающимся деревом наследования возникает через какое-то время, а именно когда новый класс будет вставлен в дерево между производным классом и его прежним суперклассом. Если придется вставить такой класс после создания непосредственного вызова суперкласса из производного класса, как показано в листинге 22.6, то вы больше не будете вызывать настоящую функцию суперкласса, а вместо этого будете вызывать реализацию суперсуперкласса. И очень часто это может быть вовсе не тем, что вам нужно.

К сожалению, компилятор не предоставляет никакой защиты против этого; этот риск неизбежен при сопровождении программы. Используйте комментарий в заголовочном файле, чтобы предупредить других программистов об этой потенциальной проблеме.

Резюме

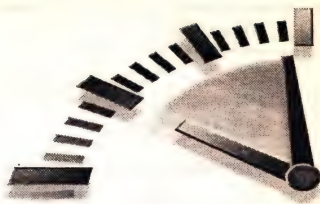
Наследование — очень мощное, но абстрактное средство. Оно в первую очередь приносит пользу программистам больших систем, которые постоянно развиваются. Как неоднократно подчеркивалось в этой книге, разнообразные расширения системы, ее восстановление, улучшение (рефакторинг) и переразложение на классы относятся к наиболее важным действиям, выполняемым программистами. Время и усилия, потраченные на эти мероприятия, значительно превосходят время и усилия, потраченные на новые программы и системы. Так что наследование играет ключевую роль в уменьшении риска введения ошибок во время сопровождения и помогает значительно уменьшить усилия, необходимые для добавления новых возможностей.

Вы научились создавать классы, производные от суперклассов, вы узнали, что ссылка на суперкласс может закончиться вызовом функций производного класса, если указано ключевое слово `virtual` (виртуальный). Это свойство называется полиморфизмом класса. Вы знаете, когда деструкторы должны быть виртуальными и как ключевое слово `virtual` (виртуальный) может помочь даже член-функции суперкласса вызвать версию производного класса член-функции суперкласса. Вы также изучили разновидности переопределения функций.

Вы видели наследование в действии и видели диаграмму дерева наследований. Чтобы полностью овладеть основными возможностями C++, вам осталось изучить всего лишь несколько тем.

УРОК 23

Испытание объектов с помощью наследования



В этом уроке вы научитесь проверять программы на C++ с помощью классов и наследования.

Написание инструментальных средств тестирования

Вспомните, что в реализации `SelfTest()` создавался экземпляр `anAccumulator` и затем он проверялся. Такие средства называются *инструментальными средствами тестирования*. Программист может и должен создавать инструментальные средства тестирования для каждого реализуемого им класса.

Инструментальное средство тестирования может быть программой, частью объекта, самим объектом. Часто оно будет иметь интерфейс пользователя. Инструментальные средства тестирования можно разделить на следующие категории.

- Всесторонние — подобные `SelfTest()` — инструментальные средства тестирования управляют всеми испытаниями и сверяют результаты с ожидаемыми. Всесторонние инструментальные средства тестирования особенно хороши для быстрого проведения регрессивного тестирования. Некоторые всесторонние инструментальные средства тестирования управляются внутрен-

ним кодом, другие же — внешними файлами или базами данных.

- Автоматизированные инструментальные средства тестирования проводят все испытания, но человек должен проверить результаты.
- Неавтоматизированные инструментальные средства тестирования предлагают все необходимые команды для управления испытаниями, но человек должен решить, какие тесты нужно запускать, в каком порядке и как интерпретировать результаты. Программу самого калькулятора можно рассматривать как неавтоматизированное инструментальное средство тестирования для класса `SAMSCalculator::aController` и используемых им объектов.

Испытание классов с помощью заранее подготовленных тестов

Испытания также подразделяются на несколько категорий.

- *Испытания с пустыми данными (empty tests)* определяют, правильно ли работает класс, программа или система при первом запуске или при отсутствии данных. Например, проверка самой последней реализации калькулятора с пустым файлом ленты показывает, что в `aPersistentTapeExternalInterface` необходимо следующее изменение:

```

1: char OperatorChar;
2: myTapeSourceInputStream >> OperatorChar;
3:
*4: if (OperatorChar == '\0')
5: {
6:     myTapeSourceInputStream.close();
7:     return anExternalInterface::GetOperatorChar();
8: }
9: else
10: {
11:     return OperatorChar;
12: };

```


Испытание с пустыми данными показывает потребность в строке 4; пустой файл возвращает символ '\0', и когда этот символ попадает в систему, он вызывает исключение из-за недопустимого оператора. Добавление условного оператора предотвращает возникновение этой проблемы.

- *Испытания с выходом за допустимые границы (out-of-bounds tests)* определяют, может ли класс обрабатывать результаты, которые находятся в пределах ограниченных типов данных параметров член-функций, но находятся вне ожидаемого набора входных данных для член-функций.
- *Испытания в условиях недостатка ресурсов (capacity tests, или stress tests)* определяют, как класс обрабатывает ситуации типа исчерпания памяти. Например, генерируется ли разумное сообщение об ошибках перед остановом испытания операционной системой, или система “бесится и сходит с ума”? Вы можете использовать методы типа повторения оператора new для выделения большого объема памяти. Это, конечно, “нагружает” вашу систему или программу, но учтите, что такие “напряженные” испытания могут быть опасны, поскольку иногда они вызывают ошибку, которая может разрушить операционную систему или даже повредить структуру файлов на диске. Лучше делать такие испытания на машине, специально выделенной для этой цели.
- *Испытания производительности (или эффективности) (performance tests)* определяют, как быстро выполняется программа и что замедляет ее выполнение. Инструментальные средства тестирования могут отображать время начала и завершения каждого испытания. Иногда отдельные испытания выполняются настолько быстро, что одно и то же испытание нужно выполнить тысячи раз, чтобы получить значимое число (в этом случае полезны циклы for). Испытания производительности могут также определять профиль выполнения программы. Например, как быстро деградирует (снижается) производительность из-за удлинения ленты? Влияет ли наиболее существенно размер ленты на запуск и завершение работы? При каких условиях он заметно влияет на ско-

рость выполнения обычных операций? Поскольку C++ часто используется в приложениях, время выполнения которых критично, испытания производительности могут играть очень важную роль.

- Имейте в виду, что даже опытные программисты не могут угадать, что замедляет работу их программ. Испытания производительности действительно помогают выяснять, что именно замедляет работу хорошо структурированной программы.
- Разработаны специальные инструментальные средства, называемые *профилировщиками*, которые позволяют определить, на что программа тратит время. Такие программы могут генерировать отчеты по классам, функциям или даже по строкам.
- Вы можете также профилировать программу, регистрируя в потоках `ofstream` или `cout` время начала выполнения различных разделов программы. Обычно для этой цели нужно использовать таймер с очень высоким разрешением. Поскольку в стандартных библиотеках такого таймера нет, приходится искать библиотеку классов от независимых разработчиков, в которой есть нужные средства.
- Чтобы локализовать проблемы с быстродействием, часто может пригодиться также метод деления пополам, или метод дихотомии (см. урок 14, “Испытание, или тестирование”). Независимо от того, используете вы профилировщик или ваши собственные инструкции для профилирования, начните с классов самого высокого уровня, сконцентрируйте усилия на самой медленной функции, профилируйте ее и повторяйте эту процедуру, пока не изолируете ту часть программы, которая является самой медленной.
- *Испытания в допустимых пределах (within-bounds tests)* позволяют удостовериться, что объект может обрабатывать ожидаемые (допустимые) входные данные. Иногда он этого делать не может. При этом виде испытаний для обнаружения проблем наиболее полезны случайно выбранные входные данные.

- *Испытания с граничными значениями (boundary value tests)* проверяют, имеет ли объект проблему с входными данными, которые, хотя и допустимы, являются граничными с точки зрения приемлемости. Как и при испытаниях с пустыми данными, при испытаниях этой категории часто обнаруживаются ошибки, которые без таких испытаний проявились бы только в процессе эксплуатации программы.

Регрессивные испытания

Чтобы создать калькулятор — вполне развитую объектно-ориентированную программу, — пришлось несколько раз выполнять рефакторинг (улучшение программы методом переразложения на классы). И каждый раз приходилось выполнять регрессивные испытания. Регрессивные испытания калькулятора, по всей вероятности, состояли из некоторого набора запросов, в котором используются все операторы. Но профессионалы обычно выполняют установленный набор тестов после каждого переразложения на классы, устранения проблемы или добавления новых возможностей.

Запись в файлы входных и выходных данных для испытаний

Инструментальные средства тестирования могут упростить процесс создания регрессивных тестов.

Неавтоматизированные инструментальные средства тестирования можно применить для подготовки файлов, используемых инструментальными средствами всестороннего тестирования. Неавтоматизированные инструментальные средства тестирования могут запрашивать входные данные, записывать их, получать результаты и выдавать подсказку для подтверждения, что результаты являются правильными. Если результаты правильны, входные данные и результаты могут быть сохранены в отдельных файлах, которые затем могут многократно читаться и использоваться в любое время инструментальными средствами всестороннего тестирования. Таким образом, неавтоматизированные инструментальные

средства тестирования и инструментальные средства всестороннего тестирования дополняют друг друга.

Использование наследования

Производный класс может использоваться для проверки суперкласса. Производный класс может быть создан исключительно для того, чтобы содержать специальные методы проверки общедоступных и защищенных методов суперкласса. Кроме того, полиморфизм класса позволяет общим инструментальным средствам тестирования проверить не только суперкласс, но и его подклассы — по крайней мере, те возможности, чьи функции объявлены в суперклассе.

Предположим, например, что у нас есть следующее наследование:

```

1: class aFirstclass                                // класс
2: {
3:     ...
4: };
5:
6: class aSecondClass: public aFirstclass           // класс
7: {
8:     ...
9: };
10:
11: class aTestHarness                               // класс
12: {
13:     bool Test(aFirstClass &theFirstClass);
14: };

```

Тогда `TestHarness::Test()` можно использовать следующим образом:

```

1: aFirstclass A;
2: aSecondClass B;
3: aTestHarness TestHarness;
4: cout << "Test of A: " << TestHarness.Test(A) << endl;
5: cout << "Test of B: " << TestHarness.Test(B) << endl;

```

Если в `aSecondClass` переопределить некоторые функции `aFirstclass`, то переопределенные функции можно будет с успехом проверить функцией `Test()`. Кроме того, можно создать потомка `aTestHarness` со второй функцией `Test()`, которая принимает в качестве параметра ссылку на `aSecondClass` вместо ссылки на `aFirstclass`. Такая член-функция может использовать `Test()` из суперкласса для проверки той

части `aSecondClass`, которая принадлежит `aFirstClass`, и может иметь дополнительный код для проверки всех дополнительных возможностей, добавленных в `aSecondClass`.

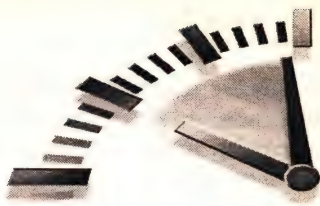
Резюме

Испытания — критическая часть разработки программ. Объекты могут свободно обращаться к другим объектам. Кроме того, они (объекты) имеют высокую степень связности и поэтому представляют собой естественную единицу (модуль), подвергаемую испытаниям. Можно проверять отдельные объекты или сборки объектов, и, кроме того, есть много различных типов испытаний и инструментальных средств тестирования.

Процесс профессионального программирования часто имеет гарантийный период, в течение которого непрограммисты выполняют регрессивные испытания разработанных объектов. Конечно, этого не достаточно. Каждый программист должен иметь опыт создания инструментальных средств тестирования и рассматривать создание и использование их как неотъемлемую часть любого программного проекта.

УРОК 24

Абстрактные классы, множественное наследование и статические члены



В этом уроке вы научитесь создавать абстрактные классы, чтобы заставить производные классы иметь некоторые члены. Вы изучите наследование от нескольких суперклассов, а также научитесь предоставлять услуги класса без создания его экземпляра.



Приступаем к изучению наиболее сложных тем по C++

Обратите внимание, что материал этого урока рассматривается в курсах для опытных программистов, использующих язык C++. Если вы тщательно проработали материал предыдущих уроков, вы уже знакомы с базовыми концепциями, необходимыми для овладения новыми. Обязательно еще раз внимательно просмотрите материал урока 22, "Наследование", перед тем, как приступить к настоящему уроку.

Создание интерфейсов

Некоторые классы имеют весьма объемную реализацию. Они по существу автономны и подобны небольшим отдельным программам, а также имеют богатый набор возможностей. Примером такого класса может быть `anAccumulator`.

Другие классы предоставляют основу реализации для их потомков, чтобы потомки могли расширять возможности суперкласса любыми способами. Примером такого класса может быть `ATare`.

Тем не менее, есть и такие классы, которые разрабатываются для того, чтобы усилить виртуальные функции так, чтобы конкретная возможность могла быть расширена или изменена. Примером такого класса может быть `anExternalInterface`. Он обращается к тем своим член-функциям, которые переопределены или расширены в классе-потомке.

Наконец, некоторые классы объявляют функции, но не реализуют их, оставляя реализацию их потомкам. Такие классы, часто называемые *интерфейсами* или *абстрактными классами*, программисты могут рассматривать как каркасы, которые указывают, что должно быть реализовано в их потомках.

Давайте рассмотрим абстрактные классы более подробно.

Чистые виртуальные функции

Возможно, термин *чистая виртуальная функция* и звучит несколько необычно, но такие функции действительно необходимы для создания абстрактных классов. Они позволяют объявить функции без реализации.

Давайте начнем с “виртуальной” части. В материалах урока 22, “Наследование”, уже говорилось, что ключевое слово `virtual` (виртуальный) позволяет при вызове функции найти соответствующую реализацию даже тогда, когда для вызова используется ссылка на суперкласс. Компилятор вызывает виртуальную функцию косвенно через указатель на функцию, который хранится в объекте.

Виртуальную функцию можно сделать “чистой” (т.е. “незагрязненной какой бы то ни было реализацией”), инициализируя ее указатель на функцию нулем. Иными словами, чтобы сделать виртуальную функцию “чистой”, достаточно обнулить указатель на функцию:

```
virtual char GetOperatorChar(void) = 0;
```

Чистая виртуальная функция обычно называется *абстрактной*. По этой причине любой класс с одной или несколькими чистыми виртуальными функциями называется *абст-*

рактным классом. Создать объект (экземпляр) из абстрактного класса нельзя, потому что программа разрушилась бы, если бы вы вызвали любую из ее чистых виртуальных член-функций. При попытке создать экземпляр абстрактного класса компилятор сгенерирует сообщение об ошибке.

Если в классе есть чистая виртуальная функция, то такой класс обладает двумя свойствами.

- Нельзя создать объект (экземпляр) этого класса, а значит, необходимо сначала создать его производный класс — потомок данного класса.
- Чтобы создать экземпляр класса-потомка, все чистые виртуальные функции в потомке необходимо переопределить.

Полиморфизм класса позволяет обращаться с объектом так, как с экземпляром его суперкласса, и потому он позволяет использовать указатели и ссылки, которые как будто бы обращаются к экземплярам абстрактных классов. Но они должны фактически указывать или ссылаться на экземпляры *конкретных* классов-потомков. Вызовы член-функций, объявленных в абстрактном классе, в конце концов будут обращениями к реализациям их конкретных классов в результате того, что данные функции являются виртуальными.



Конкретные классы

В противоположность абстрактному классу, конкретный класс не имеет никаких чистых виртуальных член-функций — ни в его собственном объявлении, ни унаследованных от его суперклассов. Большинство классов — конкретные классы. Класс, чей суперкласс абстрактен, будет конкретным только тогда, когда он реализует каждую чистую виртуальную функцию суперкласса.

Объявление абстрактного класса

Часто бывает полезно создать абстрактные классы в корне дерева наследований. Это позволяет другим программистам создавать классы, работающие с вашим абстрактным классом

даже в том случае, если вы не создали ни одного конкретного класса-потомка.

В большом проекте абстрактные классы упрощают работу одновременно над различными частями системы и впоследствии облегчают *объединение* всех классов в полноценную *интегрированную* программу. Абстрактные классы позволяют использовать компилятор для того, чтобы гарантировать, что не будет никаких несоответствий в именах функций и сигналах, заданных в вызывающей и вызываемой программах, даже тогда, когда вызывающую и вызываемую программы разработывают разные программисты.

Хотя и немного поздновато, но и в нашем программном проекте вы можете создать абстрактные классы для калькулятора.

Рассмотрим, например, `anAccumulator`, в настоящее время единственный тип сумматора в нашей системе. Чтобы создать абстрактный класс, который мог бы представлять любой тип сумматора, можно переразложить на классы первоначальный класс и сделать `anAccumulator` абстрактным. Реализация сумматора тогда может быть перемещена в класс-потомок, назовем его `aBasicAccumulator`. В листинге 24.1 показан новый заголовок для `anAccumulator`.

Листинг 24.1. Заголовок `anAccumulator` — объявление абстрактного класса

```

1: #ifndef AccumulatorModuleH
2: #define AccumulatorModuleH
3:
4: #include "RequestModule.h"
5:
6: namespace SAMSCalculator           // пространство имен
7: {
8:     class anAccumulator              // класс
9:     {
10:     public:                          // общедоступный
11:
12:         anAccumulator(void);
13:         anAccumulator(anAccumulator
14:                        theAccumulator);
15:         virtual ~anAccumulator(void);
16:         virtual float Apply
17:         (const aRequest &theRequest) = 0;
18:         virtual float Value(void) const = 0;

```

```

18:
19:         virtual anAccumulator
19:         &ReferenceToANewAccumulator(void) = 0;
20:
21:     protected:                                // защищенный
22:
23:         float myValue;
24:     };
25: };
26:
27: #endif

```

Анализ

В строках 12 и 13 объявлен конструктор и конструктор копии. Конструкторы необходимы, потому что в `anAccumulator` все еще объявлено поле `myValue` в строке 23. Конструкторы инициализируют `myValue`. Среди член-функций этого класса только конструкторы имеют реализацию.

`myValue` была перемещена в защищенный (`protected`) раздел, поэтому классы-потомки, в которых будут реализованы абстрактные функции класса `anAccumulator`, смогут использовать `myValue`. Помните, что даже классы-потомки не могут обращаться к частным (`private`) членам.

Классы-потомки смогут воспользоваться тем, что `anAccumulator` при инициализации делает `myValue` равной 0.

Строка 14 — виртуальный деструктор. Каждый класс, у которого будут наследники, должен иметь виртуальный деструктор.

Строки 16 и 17 — функции сумматора, теперь они изменены и представляют собой чистые функции, так как в начале прототипа добавлено ключевое слово `virtual` (виртуальный), а в конце — `= 0`.

В строке 19 определена новая функция, `ReferenceToANewAccumulator()`. Эта функция возвращает ссылку на объект абстрактного класса `anAccumulator`. Позже в этом уроке вы увидите, как все это работает (и почему оно нам нужно).

Реализация абстрактного класса

То, что раньше было классом `anAccumulator`, теперь реализуется классом-потомком `aBasicAccumulator`, показанным в листинге 24.2. Заголовок класса `aBasicAccumulator`

выглядит так же, как и для `anAccumulator`, за исключением того, что была удалена переменная `myValue`, но была добавлена функция `ReferenceToANewAccumulator()`. Эти изменения необходимы, потому что `myValue` является членом суперкласса и потому что `ReferenceToANewAccumulator()` должна быть реализована в `aBasicAccumulator`, чтобы вернуть его в конкретный класс.

Листинг 24.2. Заголовок `aBasicAccumulator` теперь представляет реализацию `anAccumulator`

```

1: #ifndef BasicAccumulatorModuleH
2: #define BasicAccumulatorModuleH
3:
4: #include "AccumulatorModule.h"
5: #include "InstanceCountableModule.h"
6:
7: namespace SAMSCalculator           // пространство имен
8: {
9:     class aBasicAccumulator: public anAccumulator
                                // класс
10:    {
11:        public:                // общедоступный
12:
13:            aBasicAccumulator(void);
14:            aBasicAccumulator(aBasicAccumulator
                                &theAccumulator);
15:
16:            float Apply(const aRequest &theRequest);
17:            float Value(void) const;
18:
19:            anAccumulator
                &ReferenceToANewAccumulator(void);
20:    };
21: };
22:
23: #endif

```

Изменения в `aController`

Класс `anAccumulator` теперь стал абстрактным, и у него не может быть экземпляров, так что в `aController::SelfTest()` следующая инструкция

```
anAccumulator TestAccumulator;
```

станет причиной генерации компилятором сообщения об ошибке.

Чтобы в `aController` учесть использование различных типов сумматоров в будущем, функция `aController::SelfTest()` должна получить свой экземпляр класса `anAccumulator` как результат вызова член-функции `ReferenceToANewAccumulator()`; чтобы все было правильно, этот вызов следует записать как вызов метода для своего поля `myAccumulator`:

```

anAccumulator &TestAccumulator =
↳ myAccumulator.ReferenceToANewAccumulator();

```

Если бы `SelfTest()` просто создавала и проверяла экземпляр `aBasicAccumulator`, пришлось бы изменять `SelfTest()` каждый раз, когда вы решили использовать новый класс сумматора в `aController`. Использование `ReferenceToANewAccumulator()` возлагает ответственность за предоставление нужного класса сумматора на конкретного потомка `anAccumulator`, который передается этому экземпляру `aController`, и изолирует `aController` от изменений типа передаваемого ему сумматора.

Вызов `ReferenceToANewAccumulator()` показан в листинге 24.3.

Листинг 24.3. Функция `SelfTest`, получающая экземпляр

```

aBasicAccumulator OT
myAccumulator.ReferenceToANewAccumulator()

```

```

1: void aController::SelfTest(void) const
2: {
*3:     anAccumulator &TestAccumulator =
*3:↳     myAccumulator.ReferenceToANewAccumulator();
4:
5:     try
6:     {
7:         if
8:         (
9:             TestOK(TestAccumulator,aRequest
9:↳             (aRequest::add,3),3) && // добавить
10:             TestOK(TestAccumulator,aRequest
10:↳             (aRequest::subtract,2),1) &&
                                     // вычесть
11:             TestOK(TestAccumulator,aRequest
11:↳             (aRequest::multiply,4),4) &&
                                     // умножить
12:             TestOK(TestAccumulator,aRequest

```

```

12:  (aRequest::divide,2);2)
                                     // делить
13:  )
14:  {
15:      cout << "Test OK." << endl; // все хорошо
16:  }
17:  else
18:  {
19:      cout << "Test failed." << endl;
                                     // неудача
20:  };
21:  }
22:  catch (...)
23:  {
24:      // неудача из-за исключения
25:      cout << "Test failed because of an exception.";
26:  };
27:  delete &TestAccumulator; // удалить
28: };

```

Анализ

В строке 3 `ReferenceToANewAccumulator()` вызывается для получения `TestAccumulator` от объекта (член-переменной) `myAccumulator`.

Функция `SelfTest()` работает со ссылкой на `anAccumulator`. Если бы действительно можно было получить экземпляр `anAccumulator`, все бы рухнуло при первом же вызове член-функции `TestAccumulator`. Но, хотя это и неизвестно для `SelfTest()`, `ReferenceToANewAccumulator()` фактически возвращает ссылку на экземпляр `aBasicAccumulator`, размещенный в динамической памяти, — именно этот экземпляр функция `SelfTest()` может использовать без каких-либо опасений. Все это имеет счастливый конец благодаря полиморфизму класса и виртуальным функциям.

В строке 27 освобождается память, занятая для `TestAccumulator`.

Фабрика объектов

Итак, как же все это работает? Вот как `aBasicAccumulator` реализует `ReferenceToANewAccumulator()`:

```

1: anAccumulator &aBasicAccumulator::
1:  ReferenceToANewAccumulator(void)
2:  {
3:      return *(new aBasicAccumulator);
4:  };

```

Анализ

Строка 3 создает образец `aBasicAccumulator` с помощью `new` и затем разыменовывает полученный указатель в инструкции `return` — в этом нет ничего необычного при возвращении ссылки на объект.

Возвращается ссылка на экземпляр `aBasicAccumulator`, а не на экземпляр `anAccumulator`. Но поскольку `aBasicAccumulator` — потомок `anAccumulator`, компилятор не “жалуется”. Это очередной пример, когда полиморфизм класса все упрощает.

Метод `ReferenceToANewAccumulator()` — пример *объектно-ориентированного образца (шаблона) программирования*, называемого *фабрикой*. *Объектно-ориентированные образцы* (часто называемые “ОО-образцы” или “ОО-шаблоны”) — очень модная тема среди программистов, использующих объектно-ориентированный подход на всех языках. Уже появилось несколько книг, в которых описаны различные разработанные образцы.

Оказывается, образец фабрики — это схема, с помощью которой абстрактный класс как бы генерирует свои экземпляры, хотя на самом деле, конечно, вместо этого свои экземпляры создает конкретный класс-потомок.

Абстрактные классы в дереве наследований

Абстрактные классы обычно являются корневыми классами в деревьях наследований, как показано на рис. 24.1.

Благодаря абстрактным классам команда программистов, разрабатывающих классы, может работать отдельно над классами, необходимыми для создания программы, например такими, как `SAMSCalculator`. Даже если ваши классы зависят от абстрактных классов, реализация которых поручена другим программистам, вы можете создать и скомпилировать вашу программу, используя только абстрактные классы, доступ к которым у вас был еще в начале вашего проекта. Конечно, чтобы фактически провести тестирование, вам понадобятся либо их конкретные классы, либо ваши собственные (вероятно упрощенные) конкретные классы. Такие классы часто называют *заглушками*, *куклами*, *тренажерами* или

симуляторами, и их член-функции могут всего лишь отображать сообщение, указывающее, что их вызвали, или же они могут вернуть очень простые, предопределенные (заранее запрограммированные) результаты.

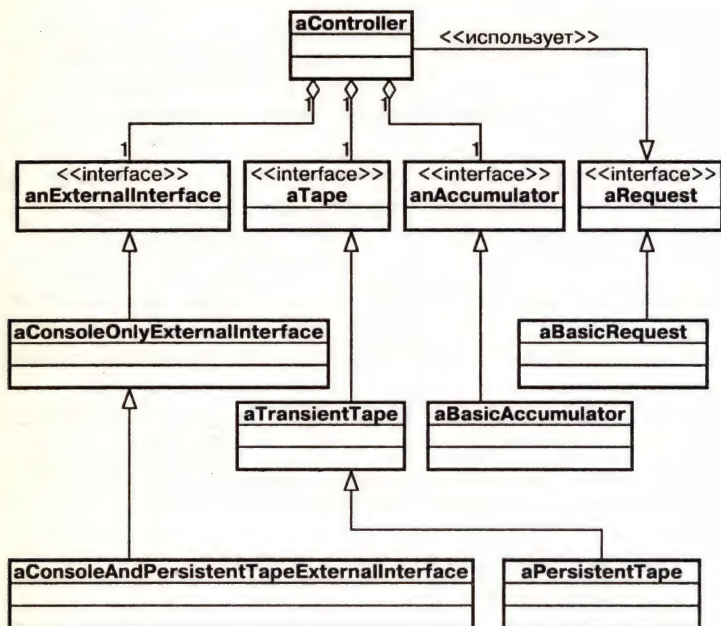


Рис. 24.1. Дерево наследований SAMSCalculator с абстрактными классами

Поскольку каждый программист, разрабатывающий классы, в конце концов создает конкретный класс, программист, выполняющий интеграцию (сборку) системы, может комбинировать эти классы с другими таким образом, чтобы получить полноценную программу. Если программисты, разрабатывающие классы, используют абстрактные классы, заглушки и инструментальные средства тестирования, то есть все шансы, что уже первый запуск после интеграции будет успешным. А так как программисты, разрабатывающие классы, могут работать по отдельности, хотя и параллельно (одновременно), программа может быть закончена в несколько раз быстрее, чем если бы один программист отвечал за все.

Множественное наследование

В C++ класс может наследовать не только от одного супер-класса, но и от нескольких суперклассов. Такое наследование называется *множественным* (или многократным) *наследованием*. Множественное (многократное) наследование — далеко не всеми одобряемое средство, которое постоянно вызывает сильные споры. Многие программисты полагают, что множественное наследование не только не нужно, но и даже опасно. Но отнюдь не меньше тех программистов, которые думают, что множественное наследование — совершенно незаменимый инструмент, и что он может использоваться вполне разумно.

Наследование представляет отношение между классами, которое часто выражается словом “является” (“is a”). Например, `aPersistentTapeExternalInterface` является `anExternalInterface` с дополнительными возможностями.

Соединение частей, или *агрегация*, — это использование объектов в качестве полей, т.е. так, как `aController` использует `anAccumulator`. Агрегация представляет отношение между экземплярами, которое часто выражается словами “имеет” (“has”) (когда экземпляр имеет члены) или “использует” (“uses”) (когда экземпляр использует члены). Неверно, что `aController` является `anAccumulator`. Он *использует* экземпляр `anAccumulator`.



Множественное наследование и соединение частей

Многие программисты используют множественное наследование тогда, когда нужно использовать соединение частей. Это — главная причина споров из-за множественного наследования. Однако при надлежащем понимании различия между отношениями “является” (“is a”) и “имеет” (“has”) (или “использует” (“uses”)), множественное наследование можно применять без каких бы то ни было опасений.

Множественное наследование *может* быть весьма полезным. Пусть, например, нужно прибавить счетчик экземпляров ко всем вашим классам, чтобы в любое время можно было выяснить, сколько объектов (экземпляров) было создано. Чтобы

сделать это, можно создать класс `isInstanceCountable`, причем сделать так, чтобы все ваши классы наследовали его. (Префикс “is” указывает, что `isInstanceCountable` добавляет возможности к существующим классам через множественное наследование.) На рис. 24.2 показано это изменение.

Чтобы применить множественное наследование, потребуются изменения в объявлении всех классов, которые должны “считать экземпляры” (“instance countable”). В листинге 24.4 показан `aBasicAccumulator` с этими изменениями.

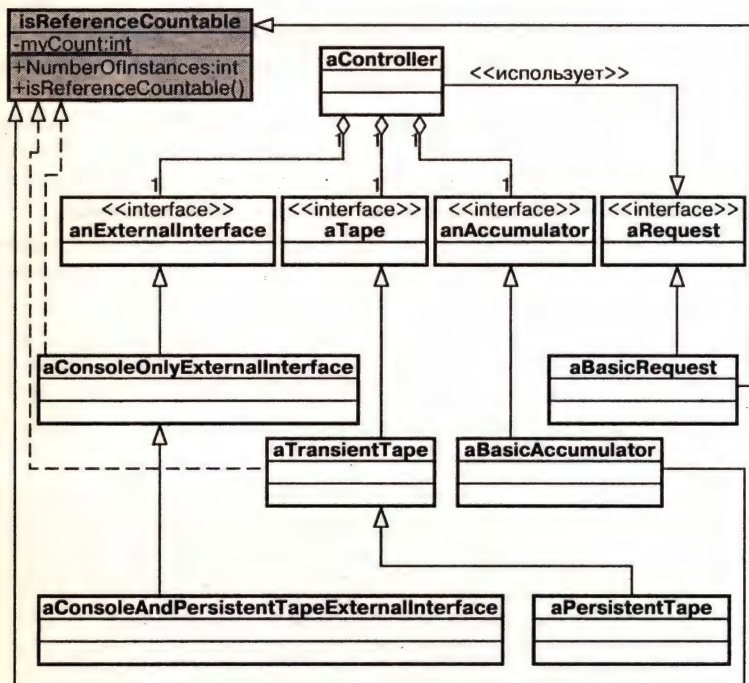


Рис. 24.2. Дерево наследований `SAMSCalculator` с абстрактными классами и множественным наследованием

Листинг 24.4. Заголовок `aBasicAccumulator`
со множественным наследованием

```

1: #ifndef BasicAccumulatorModuleH
2: #define BasicAccumulatorModuleH
3:
4: #include "AccumulatorModule.h"
*5: #include "InstanceCountableModule.h"
6:
7: namespace SAMSCalculator           // пространство имен
8: {
*9:     class aBasicAccumulator:        // класс
10:         public anAccumulator,
           public isInstanceCountable
11:     {
12:     public:
13:
14:         aBasicAccumulator(void);
15:
16:         aBasicAccumulator
17:             (aBasicAccumulator &theAccumulator);
18:
19:         float Apply(const aRequest &theRequest);
20:         float Value(void) const;
21:
22:         anAccumulator
           &ReferenceToANewAccumulator(void);
23:     };
24: };
25:
26: #endif

```

Анализ

Изменения пришлось сделать только в строках 5 (`#include` для заголовка дополнительного суперкласса) и в строках 9–10, в которых указано теперь два суперкласса. Чтобы применить множественное наследование, нужно просто указать список, каждый элемент которого состоит из модификатора доступа (в нашем примере `public` (общедоступный)) и названия (имени) класса. Такой элемент списка указывается для каждого суперкласса, причем элементы списка отделяются друг от друга запятыми.

Статические поля и функции в классах

В листинге 24.5 показана главная функция `main()`, в которой используется возможность подсчета экземпляров.

Листинг 24.5. `main.cpp` с использованием
`isInstanceCountable::`
`TotalNumberOfInstances()`

```

1: #include <iostream>
2:
3: #include
   "ConsoleAndPersistentTapeExternalInterfaceModule.h"
4: #include "BasicAccumulatorModule.h"
5: #include "PersistentTapeModule.h"
6: #include "ControllerModule.h"
7: #include "InstanceCountableModule.h"
8:
9: using namespace std; // использовать станд.
                       // пространство имен
10:
11: int main(int argc, char* argv[]) // главная программа
12: {
13:     SAMSCalculator::
13:     aConsoleAndPersistentTapeExternalInterface
14:         ExternalInterface(argv[1]);
15:
16:     SAMSCalculator::aBasicAccumulator Accumulator;
                                   // Сумматор
17:     SAMSCalculator::aPersistentTape Tape(argv[1]);
                                   // Лента
18:
19:     SAMSCalculator::aController Calculator
                                   // Калькулятор
20:     (
21:         ExternalInterface,
22:         Accumulator, // Сумматор
23:         Tape         // Лента
24:     );
25:         // Калькулятор. Работайте
26:     int ReturnCode = Calculator.Operate();
27:
28:     cout << // Общее число экземпляров всех классов
29:         "Total instances of all classes: " <<;
30:     SAMSCalculator::isInstanceCountable::

```

```

30:  TotalNumberOfInstances() <<
31:  endl;
32:
33:  char Response; // Ответ пользователя
34:  cin >> Response; // Ответ пользователя
35:
36:  return ReturnCode;
37: }

```

Анализ

В строке 26 теперь определена переменная для кода возврата из Calculator.Operate() (Калькулятор.Работайте). Это необходимо, потому что главная программа main() не возвращает управление сразу после завершения Operate(), а отображает счетчик экземпляров.

В строке 30 используется обозначение *пространство_имен::имя_класса::имя_функции*, чтобы получить счетчик всех экземпляров всех классов SAMSCalculator, существующих в данный момент.

Но как генерируется этот счетчик? И как мы можем получить счетчик экземпляров от isInstanceCountable, не имея экземпляра этого класса?

Статические (static) члены класса

Когда лента Tape() и сумматор Accumulator() были функциями, в них использовались статические переменные. Тогда статическая переменная инициализировалась при запуске программы, причем ее значение сохранялось между вызовами функций.

От такого типа переменной пришлось отказаться при повторном разложении калькулятора на классы, потому что члены-переменные класса позволяли сумматору и ленте сохранять внутреннее состояние не хуже, чем статическая переменная.

Однако статическая переменная может играть важную роль в классах. Статическая переменная в классе инициализируется при запуске программы и совместно используется всеми экземплярами класса. Такая переменная может хранить счетчик всех экземпляров всех классов SAMSCalculator.

В листинге 24.6 приведено объявление `isInstanceCountable`, в котором в специальной переменной хранится счетчик экземпляров.

Листинг 24.6. Объявление `isInstanceCountable`

```

1: #ifndef InstanceCountableModuleH
2: #define InstanceCountableModuleH
3:
4: namespace SAMSCalculator // пространство имен
5: {
6:     class isInstanceCountable // класс
7:     {
8:         public: // общедоступный
9:
10:             isInstanceCountable(void);
11:             ~isInstanceCountable(void);
12:
13:             static int TotalNumberOfInstances(void);
14:             // статическая переменная
15:         private: // частный
16:
17:             static int ourInstanceCounter;
18:             // статическая переменная
19:     };
20:
21: };
22:
22: #endif

```

Анализ

В строке 17 объявлена статическая переменная типа `int`, которая совместно используется всеми экземплярами класса `isInstanceCountable`. Ее название (имя) имеет префикс “our” (“наша”), который напоминает, что эта переменная “расположена” на уровне класса.

Если создать три объекта этого класса, `ourInstanceCounter` будет иметь значение 3. Чтобы увидеть, как это делается, нужно посмотреть на реализацию конструктора для этого класса.

Поскольку каждый класс калькулятора `SAMSCalculator` является наследником этого класса, а также его интерфейса, все классы калькулятора `SAMSCalculator` совместно используют эту переменную.

В строке 13 объявлена *статическая функция*. Ключевое слово `static` (статический) в член-функции означает, что такую функцию можно вызвать даже без экземпляра класса. Статические функции также называются *функциями класса*, потому что для их вызова используется имя класса и оператор разрешения области видимости (`имя_класса::имя_функции()`), или, как в нашем случае, `имя_пространства_имен::имя_класса::имя_функции()`). В строке 30 главная функция `main()` вызывает эту функцию.

Увеличение (инкрементирование) и уменьшение (декрементирование) счетчика экземпляров

Конструктор и деструктор `isReferenceCountable`, показанные в листинге 24.7, увеличивают (инкрементируют) и уменьшают (декрементируют) `ourInstanceCounter`.

Листинг 24.7. Реализация `isInstanceCountable`

```
1: #include "InstanceCountableModule.h"
2:
3: namespace SAMSCalculator // пространство имен
4: {
5:     int isInstanceCountable::ourInstanceCounter = 0;
6:
7:     isInstanceCountable::isInstanceCountable(void)
8:     {
9:         ourInstanceCounter++;
10:    };
11:
12:    isInstanceCountable::~isInstanceCountable(void)
13:    {
14:        ourInstanceCounter--;
15:    };
16:
17:    int InstanceCountable::TotalNumberOfInstances(void)
18:    {
19:        return ourInstanceCounter;
20:    };
21:
22: };
```

Анализ

Строка 5 инициализирует статическую переменную-счетчик `ourInstanceCounter`. Это происходит при запуске программы и не связано с каким-либо экземпляром класса. Конструкция в строке 5 называется *статической инициализацией*.

Строки 7–10 увеличивают счетчик, когда создается экземпляр этого класса. Благодаря множественному наследованию при создании экземпляра класса-потомка вызываются конструкторы всех его суперклассов, и потому при создании экземпляра любого потомка `isInstanceCountable` увеличивается `ourInstanceCounter`.

Строки 12–15 декрементируют (уменьшают) счетчик, когда уничтожается экземпляр этого класса. Как и конструкторы при создании объектов, при разрушении производного класса вызываются деструкторы суперклассов.

В результате счетчик увеличивается при создании экземпляра каждого класса и уменьшается при его разрушении.

**Помните, что деструкторы должны быть виртуальными**

Счетчик экземпляров `isInstanceCountable` не будет работать должным образом, если деструкторы классов `SAMSCalculator` не будут виртуальными. Для каждого интерфейсного класса нужно объявить виртуальный деструктор. Это необходимо для того, чтобы при разрушении классов через ссылку на абстрактный класс (как это делается в `SelfTest()`) был вызван правильный деструктор, а не просто деструктор абстрактного класса.

Вывод главной программы `main.cpp`

Ниже представлен результат выполнения модифицированного калькулятора. Подсказок теперь нет, поскольку калькулятор только считывает ввод, но подсказок не выводит:

ВВОД

+3-2*4/2=

ВЫВОД

2

ВВОД

!

ВЫВОД

+ - ОК.
- - ОК.
* - ОК.
/ - ОК.
Test ОК.

ВВОД

ВЫВОД

Total instances of all classes: 8
(Всего экземпляров всех классов: 8)

Правильно ли все это работало? Если вы посмотрите на главную программу `main()`, вы убедитесь, что создаются экземпляры четырех классов: `aController`, `aBasicAccumulator`, `aConsoleAndPersistentTapeExternalInterface` и `aPersistentTape`. Итак, мы нашли четыре из восьми экземпляров, упомянутых в отчете.

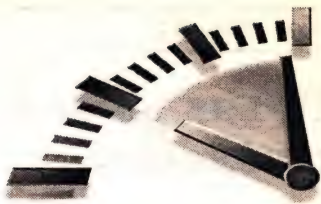
Кроме того, пользователь проводит четыре вычисления, каждое из которых генерирует один экземпляр `aRequest` для ленты, — всего создается еще четыре экземпляра, и потому общее количество увеличивается до восьми. Также обратите внимание, если бы вы не объявили виртуальный деструктор в `anAccumulator`, счетчик был бы равен 9, потому что деструктор `isInstanceCountable` не вызывался бы, когда `SelfTest()` удаляла `TestAccumulator`. Помните, `isInstanceCountable` унаследован классом `aBasicAccumulator`, а не его суперклассом `anAccumulator`.

Резюме

Вы изучили очень сложный материал по C++, включая объявление чистых абстрактных функций, абстрактные и конкретные классы, роль интерфейсов в дереве наследований, использование множественного наследования и типичные ошибки при его использовании, а также роль статических переменных и функций в классах.

УРОК 25

Шаблоны



В этом уроке вы научитесь объявлять и определять шаблоны. Шаблоны позволяют формировать классы, которые работают с разнообразными типами, причем нет необходимости переписывать класс для каждого отдельного типа.

Сила и слабость шаблонов

Иногда трудно выбрать тип данных или класс, который составит ядро вашей программы или класса.

Например, в калькуляторе использовался тип `int`, пока не были выявлены проблемы арифметического характера, связанные с таким выбором. В уроке 4, “Ввод чисел”, нам пришлось изменить калькулятор так, чтобы использовать тип `float` вместо `int`. По этой причине пришлось модифицировать почти весь код.

Изменение объектно-ориентированного калькулятора в целях увеличения точности и, как следствие, использования чисел двойной точности (`double`) вместо чисел с плавающей точкой (`float`) не привнесло бы ничего нового: оно потребовало бы изменений в каждом классе. А если бы вы захотели иметь калькулятор, работающий с числами типа `int`, калькулятор, работающий с числами типа `float` (с плавающей точкой), и калькулятор, работающий с числами типа `long double`, — и притом все сразу (т.е. в то же самое время), вам пришлось бы повторно, притом несколько раз, переписать реализацию классов, используемых калькулятором.

Шаблоны позволяют избежать этих далеко идущих изменений и множественных реализаций — благодаря им понадобится всего лишь одна реализация.

Объявление и использование шаблонов

Вы уже использовали шаблон при изучении материала урока 20, “Остальные классы калькулятора”, в `aTape: std::vector<aRequest> myTape;`. Это объявление поля создавало экземпляр вектора, в который можно было записывать только объекты `aRequest`.

Такие шаблоны можно представлять в виде формы (или бланка), по которой компилятор генерирует код для класса, когда вы “заполните пробелы”.

Шаблоны обычно имеют только один (в крайнем случае, несколько) таких “пробелов”, которые называются *параметрическими типами*. Значение, указываемое для параметрического типа, заменяет данный параметрический тип везде, где он встречается в шаблоне. В `std::vector<aRequest>` в качестве значения параметрического типа шаблона вектора `vector` указано `aRequest`.

Формат для кодирования шаблона может казаться несколько сложным, но на самом деле он довольно прост. Как и любой класс, шаблон класса имеет две основные части: объявление и определение.

В объявлении шаблона к началу обычного объявления класса прибавляется конструкция вида `template <класс параметрический_тип>` (*шаблон <класс параметрический_тип>*). Она идентифицирует название (имя) параметрического типа и сообщает компилятору, что класс является классом шаблона¹.

В определении шаблона (реализации класса шаблона) к началу каждого заголовка определения функции-члена прибавляется конструкция вида `template <класс параметрический_тип>` (*шаблон <класс параметрический_тип>*), причем в заголовке определяемой функции после имени класса следует конструкция `<параметрический_тип>`, которая сообщает компилятору, что функция является членом класса шаблона.

¹ Потому что он входит в состав шаблона. Весь шаблон класса также очень часто называют *классом шаблона*, *шаблонным классом*, *параметрическим классом* и даже *параметрическим типом*. — Прим. ред.

Вот пример заголовочного файла шаблона, где **ofType** (Типа) — параметрический тип:

```

1: #include <filename>
2: namespace SomeNamespace
3: {
4: // Шаблон <класс Типа> класс aSampleTemplate
5:     template <class ofType> class aSampleTemplate
6:     {
7:         public:
8:             ofType SomeFunction
9: // Параметр типа ofType theOtherArgument
10:             (int theArgument, ofType
                                     theOtherArgument);
11:     };
12:
13: // Шаблон <класс ofType> ofType
14:     template <class ofType> ofType
15:         aSampleTemplate<ofType>::SomeFunction
16:         (int theArgument, ofType theOtherArgument)
17:     {
18:         int Thing = theArgument;
19:         ofType OtherThing = theOtherArgument;
20:         return OtherThing;
21:     };
22: };

```



Помещайте весь код шаблона в заголовочном файле

Реализация класса шаблона должна быть закодирована в заголовочном файле, а не в файле реализации (.cpp), в противном случае компилятор или компоновщик сгенерирует сообщения об ошибках.

Если класс находится в пространстве имен, то это пространство имен должно начинаться перед объявлением класса и закончиться не ранее конца последней функции в реализации.

Чтобы инструкции #include, включающие файлы, необходимые реализации, использовались правильно, они должны быть помещены в заголовок для шаблона.

Когда вы с помощью этого шаблона определяете переменную с плавающей точкой, указывая float в качестве значения параметрического типа ofType (Типа), например, вот так:

`aSampleTemplate<float> SampleThing; // <с плавающей точкой>`
компилятор внутренне генерирует объявление класса и реализацию, основанные на обеспеченном типе. Это называется специализацией.



Специализация

Создание экземпляра (объекта) по шаблону. Говорят, что шаблон специализируется, когда для параметрического типа указывается его фактический тип. Например, `std::vector <aRequest>` — класс шаблона `std::vector` (вектор), специализированный для типа `aRequest`.

Давайте представим, на что мог бы быть похож такой сгенерированный класс, если бы мы могли его увидеть. (На самом деле, и класс, и его название (имя) генерируются компилятором скрытно, и в распечатке нигде не отображаются.) Если предположить, что имя класса состоит из значения параметрического типа, за которым следует имя класса шаблона, то в нашем случае названием (именем) сгенерированного класса было бы `floataSampleTemplate`. А сам класс выглядел бы примерно так:

```
1: #include <filename>
2: namespace SomeNamespace           // пространство имен
3: {
4:
5:     class floataSampleTemplate      // сгенерированный
                                     // класс
6:     {
7:         public:
8:             float SomeFunction      // с плавающей точкой
9:                                     // Параметр
                                     // с плавающей точкой
10:            (int theArgument, float theOtherArgument);
11:    };
12:
13:
14:
15:    aSampleTemplatefloat::SomeFunction
16:    (int theArgument, float theOtherArgument)
17:    {                                // Параметр
                                     // с плавающей точкой
18:        int Thing = theArgument;
```



```

19:         float OtherThing = theOtherArgument;
20:         return OtherThing;
21:     };
22: };

```

Этот класс никак нельзя увидеть в коде — компилятор генерирует класс тогда, когда транслирует вашу программу. Однако компилятор обращается со сгенерированным классом так же, как он обращался бы с любым классом, написанным программистом непосредственно в исходном тексте программы.

Определение переменной

```

aSampleTemplate<float> SampleThing; // <с плавающей точкой>

```

компилятор изменяет так, чтобы в нем использовалось внутреннее имя того класса, который был сгенерирован компилятором:

```

floataSampleTemplate SampleThing

```

Конечно же, этого нам никогда не увидеть, так как все происходит “за кулисами”.

Калькулятор как система шаблонов

Теперь давайте применим шаблоны в программе калькулятора. В листинге 25.1 показано, как объявление `aRequest` превратить в шаблон.

Листинг 25.1. `aRequest` — объявление в виде шаблонного класса

```

*1: template <class ofType> class aRequest:
    // Шаблон класс aRequest
*2: public isInstanceCountable
3: {
4:     public:
5:
6:     // Замечание: конструктора по умолчанию нет
7:     // Создавая экземпляр этого класса,
8:     // нужно указать оператор и операнд
9:
*10:         aRequest
*11:         (
*12:             const anOperator theOperator,
*13:             const ofType anOperand
*14:         );
15:
16: // Позволить копирование

```

```

17:
18:         aRequest(const aRequest &theOtherRequest);
19:
20: // Поддержка присваивания
21:
22:         aRequest &operator =
23:             (const aRequest &theOtherRequest);
24:
25: // Эти функции могут вызываться без экземпляра:
26:
27:         static char OperatorAsCharacter
28:             (const anOperator theOperator);
29:
30:         static anOperator CharacterAsOperator
31:             (const char theCharacter);
32:
33: // Нельзя изменять оператор или операнд
34: // после создания экземпляра этого класса;
35: // можно только получить их значения
36:
37:         anOperator Operator(void) const;
38:         char OperatorCharacter(void) const;
39:
40:         ofType Operand(void) const;
41:
42:     private: // частный
43:
44:         anOperator myOperator;
45:         ofType myOperand;
46: };

```

Анализ

В этом листинге показан только раздел описаний заголовочного файла. Реализация шаблона также должна быть в заголовочном файле. Это необходимо для того, чтобы компилятор смог сгенерировать на основе параметрического типа необходимую реализацию во время компиляции. Файл реализации для `aRequest` (файл `RequestModule.cpp`) теперь будет содержать только `#include` для его заголовочного файла. (Стандарт ISO/ANSI на C++ требует наличия в файле реализации хотя бы некоторого кода и `#include` для заголовка.)

Реализация класса шаблона должна немедленно следовать за объявлением класса в заголовочном файле.

Объявление перечислимого типа `anOperator` было перемещено из объявления для `aRequest`. Компилятор, который я использовал при написании этой книги, не может найти тип, вложенный в класс шаблона, так что некоторые выражения

(вроде `aRequest<float>::add`) в других местах программы вызывают ошибку. Ваш компилятор может испытывать такие же трудности с поиском объявлений типа, вложенных в шаблоны, но вполне возможно, что ваш компилятор с такими задачами справляется без каких бы то ни было проблем. Имейте в виду, что шаблоны — сравнительно новая возможность, и разные компиляторы обрабатывают их по-разному.

Строки 1 и 2 — новый заголовок объявления класса, в нем указано, что параметрический тип называется `ofType` (Типа).

В строках 10–14 объявлен конструктор класса, в нем параметрический тип используется для обозначения типа параметра `theOperand` в строке 13.

В строке 40 параметрический тип используется для обозначения типа значения, возвращаемого функцией `Operand()` (т.е. для обозначения типа операнда).

В строке 45 объявлено поле `myOperand`, и снова параметрический тип `ofType` (Типа) используется для обозначения типа этого поля.

В листинге 25.2 показано, как изменяется определение конструктора для этого класса, с учетом того, что конструктор является частью шаблона. Помните, что это определение находится внутри объявления пространства имен, в котором размещен весь заголовочный файл, и притом это определение — только одно из нескольких определений функций, которые следуют за объявлением класса.

Листинг 25.2. Конструктор `aRequest` в заголовочном файле шаблона

```
*1: template <class ofType> aRequest<ofType>::aRequest
2: {
3:     const anOperator theOperator,
*4:     const ofType theOperand
5: }:
6:     myOperator(theOperator),
*7:     myOperand(theOperand)
8: {
9: };
```

Анализ

В строке 1 указано, что данная функция использует параметрический тип `ofType` и является членом класса `aRequest<ofType>`.

В строке 4 параметрический тип замещает тип входного параметра конструктора.

Строка 7 инициализирует поле. Обратите внимание, что эта строка не изменилась, она такая же, как и в предыдущей, нешаблонной, версии конструктора.

В листинге 25.3 показана функция `Operand()` (Операнд), шаблонная версия которой только слегка отличается от нешаблонной.

Листинг 25.3. `aRequest`. Член-функция `Operand()` (Операнд) в заголовочном файле шаблона

```
*1: template <class ofType> // Шаблон - класс
*2: ofType aRequest<ofType>::Operand(void) const
3: {
*4:     return myOperand;
5: };
```

Анализ

В строке 1 указано, что данная член-функция является частью шаблона, причем в качестве названия (имени) параметрического типа указан идентификатор `ofType`.

Строка 2 — заголовок функции, причем в качестве возвращаемого типа указан параметрический тип. (Иными словами, экземпляр функции возвращает тот тип, который замещает параметрический тип.)

В строке 4 возвращается значение поля — эта строка не изменилась по сравнению с предыдущей версией программы.

Изменение `anAccumulator` и `aBasicAccumulator`

Вы, возможно, помните, что, работая над материалом урока 24, “Абстрактные классы, множественное наследование и статические члены”, мы сделали класс `anAccumulator` абстрактным, а `aBasicAccumulator` стал классом реализации. Оба класса должны стать шаблонами, чтобы они могли работать с `aRequest<ofType>`.

Поскольку класс `anAccumulator` абстрактный, его реализация очень мала, а значит, его заголовочный файл весьма прост — можете убедиться в этом сами, взглянув на листинг 25.4.

Листинг 25.4. anAccumulator. Заголовочный файл шаблона

```

1: #ifndef AccumulatorModuleH
2: #define AccumulatorModuleH
3:
4: #include "RequestModule.h"
5:
6: namespace SAMSCalculator
7: {
8:     template <class ofType> class anAccumulator
        // шаблон - класс
9:     {
10:     public:
11:
12:         anAccumulator(void);
13:         anAccumulator
            anAccumulator &theAccumulator);
14:
15:         virtual ~anAccumulator(void);
            // Виртуальный
            // Виртуальные функции
16:         virtual ofType Apply
17:             (const aRequest<ofType>
18:              &theRequest) = 0;
19:
20:         virtual ofType Value(void) const = 0;
21:
22:         virtual anAccumulator
23:             &ReferenceToANewAccumulator(void) = 0;
24:
25:     protected: // защищенный
26:
27:         ofType myValue;
28:     };
29:
30:     using namespace std; // Используем
        // пространство имен std
31:     // функции в классе шаблона
32:     template <class ofType>
33:         anAccumulator<ofType>::anAccumulator(void):
34:             myValue(0)
35:         {
36:         };
37:
38:     template <class ofType>
39:         anAccumulator<ofType>::anAccumulator
40:             (anAccumulator &theAccumulator):
41:             myValue(theAccumulator.myValue)
42:         {
43:         };

```

```

44:
*45:     template <class ofType>
*46:         anAccumulator<ofType>::~~anAccumulator(void)
*47:     {
*48:     };
49:
50: };
51:
52: #endif

```

Анализ Строка 4 — обычная команда `#include aRequest`.

В строке 8 указано, что класс является шаблоном, причем в качестве названия (имени) параметрического типа выбран идентификатор `ofType` (Типа).

Строки 32–34 — конструктор. Хотя этот конструктор не использует параметрический тип, необходимо указание на то, что это конструктор шаблонного класса, — вот зачем здесь `template <class ofType>` и `anAccumulator<ofType>`. Если бы не эти конструкции, компилятор не знал бы, что это член-функция шаблонного класса `anAccumulator<ofType>`.

То же самое справедливо и для всех остальных функций, включая виртуальный деструктор, приведенный в строках 45–48.

В листинге 25.5 показан `aBasicAccumulator`, теперь он также является шаблоном.

Листинг 25.5. `aBasicAccumulator`. Заголовочный файл шаблона

```

1: #ifndef BasicAccumulatorModuleH
2: #define BasicAccumulatorModuleH
3:
*4: #include <string>
*5: #include <exception>
6:
7: #include "AccumulatorModule.h"
8: #include "InstanceCountableModule.h"
9:
10: namespace SAMS {
11: {
*12:     template <class ofType> class aBasicAccumulator:
*13:     public anAccumulator<ofType>,
*14:         public isInstanceCountable
15:
16:     public:

```



```

17:
18:         aBasicAccumulator(void) :
19:
20:         aBasicAccumulator
21:             (aBasicAccumulator &theAccumulator);
22:
*23:         ofType Apply(const aRequest<ofType>
                        &theRequest);
24:         ofType Value(void) const;
25:
*26:         anAccumulator<ofType>
*27:             &ReferenceToANewAccumulator(void);
28:     };
29:
30:     using namespace std; // Использовать
                           // пространство имен std
31:                           // Член-функции шаблонного класса
32:     template <class ofType>
33:         aBasicAccumulator<ofType>::aBasicAccumulator
                                   void)
34:     {
35:     };
36:
37:     template <class ofType>
38:         aBasicAccumulator<ofType>::aBasicAccumulator
39:             (aBasicAccumulator<ofType> &theAccumulator):
40:             anAccumulator(theAccumulator)
41:         {
42:         };
43:
*44:     template <class ofType>
*45:         ofType aBasicAccumulator<ofType>::Apply
*46:             (const aRequest<ofType> &theRequest)
47:         {
48:             switch (theRequest.Operator())
49:             // Переключатель (Оператор)
50:             {
51:                 case add: // Случай сложения:
52:                     myValue+= theRequest.Operand();
53:                     break;
54:                 case subtract: // Случай вычитания:
55:                     myValue-= theRequest.Operand();
56:                     break;
57:                 case multiply: // Случай умножения:
58:                     myValue*= theRequest.Operand();
59:                     break;
60:                 case divide: // Случай деления:

```

```

63:             myValue/= theRequest.Operand();
64:             break;
65:
66:         default:
67:
68:             throw
69:                 runtime_error
70:                 (
71:                     string("SAMSCalculator::") +
72:                     string("aBasicAccumulator<
73:                         ofType>::") +
74:                     string("Apply") +
75:                     string(" - Unknown
76:                         operator.")
77:                 );
78:         return Value();
79:     };
80:
81:     template <class ofType>
82:         ofType aBasicAccumulator<ofType>::Value
83:             void) const
84:     {
85:         return myValue;
86:     };
87:     template <class ofType>
88:         anAccumulator<ofType>
89:         &aBasicAccumulator<ofType>::
90:         ReferenceToANewAccumulator(void)
91:     {
92:         return *(new aBasicAccumulator<ofType>);
93:     };
94: };
95:
96: #endif

```

Анализ

В строках 4 и 5 включаются заголовочные файлы, необходимые для реализации. Как упоминалось ранее, чтобы шаблон работал, все необходимое для реализации должно быть помещено в заголовочный файл.

В строках 12–14 показано, что `aBasicAccumulator` является наследником шаблонного класса `anAccumulator` и нешаблонного класса `isInstanceCountable`. В строке 13 после имени класса `anAccumulator` следует также параметрический тип в угловых скобках. Такая специализация гарантирует, что

`anAccumulator` будет иметь подходящий параметрический тип, который будет передан `aBasicAccumulator`, и что вследствие этого `myValue` также будет иметь подходящий тип. В строке 14 показано, что нешаблонные классы наследуются, как обычно, и параметрический тип на них никак не влияет.

В строке 23 объявление функции изменено с учетом параметрического типа; особо нужно отметить необходимость использования параметрического типа для специализации `aRequest` в параметре функции `theRequest`. Специализация гарантирует, что операнд запроса будет иметь тот же самый тип, что и `anAccumulator::myValue`, потому что они оба будут иметь тип, который `aBasicAccumulator` получит в качестве параметрического типа.

В строках 26 и 27 объявлена функция `ReferenceToANewAccumulator()`. В данном случае возвращаемый тип `anAccumulator<ofType>` был специализирован параметрическим типом. Это гарантирует, что новый экземпляр будет членом того же самого класса, для которого потребовался новый экземпляр.

В строках 44–46 показан заголовок функции `Apply()` (Применить), который в строке 46 специализирует `aRequest`, чтобы он получил параметрический тип, переданный для этого класса. Остальная часть функции по существу не изменилась, если не считать изменения, вызванные вынесением перечисления (`enum`) для `anOperator` из декларации класса `aRequest` для значений, используемых в случаях инструкции выбора (переключатель `switch`). (Перед этими значениями больше нет префикса `aRequest::`.)

В строках 87–95 показан новый метод `ReferenceToNewAccumulator()`. Наиболее интересна строка 92, потому что в ней используется `new` для создания экземпляра класса `aBasicAccumulator<ofType>` и возвращается ссылка на созданный экземпляр как ссылка на `anAccumulator<ofType>`. Это явная демонстрация того, что полиморфизм класса работает не только для нешаблонных классов, но и для шаблонных классов.

Использование шаблонов

В листинге 25.6 показано, как шаблоны `aRequest` и `aBasicAccumulator` используются в новой главной программе `main()`, которая выступает в роли инструментального средства тестирования для этих классов. Это инструментальное средство тестирования создает экземпляры двух отдельных сумматоров — один специализированный для `float` и один специализированный для `int`. Как вы помните из урока 4 “Ввод чисел”, целочисленный калькулятор отбрасывает дробные части чисел. Когда целочисленный калькулятор проверялся на выражении $+3-2*3/2$, он выдал в результате 1, а калькулятор, работающий с числами с плавающей точкой (`float`), в результате вычислений выдает 1.5.

Листинг 25.6. `main()`. Использование шаблонных классов в главной программе для создания экземпляров и испытание калькулятора, обрабатывающего целочисленные данные (типа `int`) и данные с плавающей точкой (типа `float`)

```

1: #include <iostream>
2:
3: #include "BasicAccumulatorModule.h"
4: #include "RequestModule.h"
5:
6: using namespace std;           // Использовать пространство
                                // имен std;
7: using namespace SAMSCalculator; // SAMSCalculator;
8:
9: int main(int argc, char* argv[]) // главная программа
10: {                               // Сумматор <с плавающей точкой>
*11:     aBasicAccumulator<float>    Accumulator;
12:
*13:     Accumulator.Apply(aRequest<float>(add,3));
                                // добавить 3
*14:     Accumulator.Apply(aRequest<float>(subtract,2));
                                // - 2
*15:     Accumulator.Apply(aRequest<float>(multiply,3));
                                // * 3
*16:     Accumulator.Apply(aRequest<float>(divide,2));
                                // / 2
17:     // " Результат = " << Сумматор.Значение
18:     cout << "Result = " << Accumulator.Value() << endl;
19:
20:     char StopCharacter; // Символ

```

```

21:     cin >> StopCharacter;
22:
*23:     aBasicAccumulator<int>    IntAccumulator;
24:
*25:     IntAccumulator.Apply(aRequest<int>(add,3));
                                   // добавить 3
*26:     IntAccumulator.Apply(aRequest<int>(subtract,2));
                                   // - 2
*27:     IntAccumulator.Apply(aRequest<int>(multiply,3));
                                   // * 3
*28:     IntAccumulator.Apply(aRequest<int>(divide,2));
                                   // / 2
29:     // "Результат = " << IntAccumulator.Значение
30:     cout << "Result = " << IntAccumulator.Value() <<
        endl;
31:
32:     cin >> StopCharacter;
33:
34:     return 0;
35: }

```

Анализ

В строке 11 объявлен сумматор <с плавающей точкой> `aBasicAccumulator<float>` `Accumulator`, специализированный на использовании чисел с плавающей точкой (типа `float`), — именно этот тип замещает параметрический тип.

Строки 13–16 тестируют этот сумматор с помощью набора объектов `aRequest`, специализированных на использовании чисел с плавающей точкой (типа `float`). Если бы вместо `aRequest<float>` для параметра использовался `aRequest<int>`, компилятор в строках 13–16 обнаружил бы ошибки и сгенерировал бы примерно такие сообщения о них:

```

[C++ Error] Main.cpp(13):
E2064 Cannot initialize 'const aRequest<float> &'
with 'aRequest<int>'
[C++ Error] Main.cpp(13):
E2342 Type mismatch in parameter 'theRequest'
(wanted 'const aRequest<float> &', got 'aRequest<int>')

```

```

[Ошибка C++] Main.cpp (13):
E2064 Нельзя инициализировать 'константа aRequest<с
плавающей точкой> &'
с помощью 'aRequest<int>'
[Ошибка C++] Main.cpp (13):
E2342 Несоответствия типов для параметра 'theRequest'
(должна быть 'константа aRequest<с плавающей точкой> &',
а на самом деле 'aRequest<int>')

```

В строках 13–16 создаются объекты (экземпляры `aRequest<float>`) в качестве параметров `Accumulator.Apply()` (метод применения сумматора для выполнения запроса). Эти объекты рассматриваются как константы, в отличие от экземпляра сумматора `Accumulator`, который создается как переменная. Как видите, как и для любых других классов, для создания экземпляров шаблонных классов можно применить любой способ.

В строках 23 и 25–28 то же самое испытание выполняется на `aBasicAccumulator` и `aRequest`, теперь уже специализированных на `int`.

Выполнение испытания

При выполнении инструментального средства тестирования получается следующий вывод:

ВЫВОД

Result = 1.5
Результат = 1.5

ВВОД

ВЫВОД

Result = 1
Результат = 1

Как видите, версия сумматора для типа `int` имеет проблему с целочисленным делением, что было продемонстрировано еще в материалах урока 4, “Ввод чисел”. Но сумматор работает так, как ему и полагается для этого типа, и тем самым он демонстрирует, что экземпляр `aBasicAccumulator`, как и требовалось, был создан для типа `int`.

Несколько замечаний о шаблонах

Ниже представлено несколько дополнительных фактов, относящихся к шаблонам.

- Шаблоны могут иметь несколько параметрических типов. Параметрические типы отделяются запятыми. Например:

```
1: template <class ofOperandType, class ofOperatorType>
2: class Something...
```


- Шаблоны могут наследовать от других шаблонов, нешаблонных классов или любого сочетания шаблонных и нешаблонных классов.
- Нешаблонные классы могут наследовать от одного или нескольких шаблонов, специализированных с помощью соответствующих параметрических типов, например:
1: `class aNonTemplateClass:public aTemplate<float>`

Сила и слабость шаблонов

Из этого урока видно, что использование шаблонов имеет определенные преимущества. Например, вы создали два различных сумматора без необходимости повторно писать реализацию главных классов — чтобы изменить тип данных сумматора, вы изменили только значение параметрического типа. Те же самые преимущества можно извлечь из программы калькулятора, если преобразовать остальные классы калькулятора в шаблоны.



Риски, связанные с шаблонами

Вся реализация шаблона находится в файле заголовка и потому фактически открыта. Поэтому шаблоны невозможно перепродавать как библиотеки независимых разработчиков (третьих лиц) без исходного текста, и по этой причине нельзя быть уверенным, что вызывающие программы не полагаются на детали реализации.

Объявление шаблона и его реализация использует довольно сложный синтаксис. Чтобы еще более усугубить эту проблему, многие разработчики шаблона используют `T`, а не что-нибудь более осмысленное в качестве идентификатора параметрического типа, а это делает код еще более загадочным.

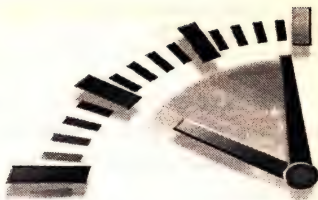
Наконец, некоторые продвинутые программисты — фанаты шаблонов — используют шаблоны сложными способами, понять которые подчас очень трудно любому человеку. Неудивительно, что по этой причине были (и еще будут) написаны толстые книги о шаблонах. Позвольте все же дать вам совет: старайтесь использовать шаблоны простыми, понятными способами.

Шаблоны очень хорошо поддерживают строгий контроль типов, что позволяет компилятору C++ предупредить об ошибках в коде. Использование параметрического типа в шаблонах означает, что шаблоны можно использовать вместе с наследованием и в параметрах член-функций, и вы можете быть уверены, что основные типы данных совместны. Компилятор выдаст предупреждение, если это не так. Иными словами, чтобы убедиться в совместности типов, достаточно просмотреть сообщения об ошибках, если таковые будут. Так что если программа компилируется успешно, то вы можете быть уверены, что типы в ней используются правильно, причем то же относится и к параметрическим типам.

Резюме

Вы научились объявлять шаблонные классы, определять их и создавать их экземпляры. Вы научились использовать шаблонные классы в качестве параметров член-функций других шаблонных классов и знаете, как использовать параметрический тип, чтобы удостовериться, что унаследованный класс и шаблонный класс в параметре специализированы так, что соответствуют классу, который их использует, — это позволяет положиться на безопасное использование типов в C++. И конечно, вы изучили преимущества и риски использования шаблонов.

УРОК 26



Эффективность: оптимизация в C++

В этом уроке обсуждаются способы ускорения выполнения программы.

Выполняем быстрее и уменьшаем объем

Каждую программу можно заставить работать быстрее или использовать меньший объем памяти. Многие программисты утверждают, что могут предсказать, что в программе выполняется быстро, а что — медленно, что занимает большой объем памяти, а что — малый. Однако опыт показывает, что программисты обычно не могут угадать заранее, в каких частях их программ возникнут проблемы с эффективностью. И часто, преждевременно пытаясь оптимизировать программу, программист создаст не только более медленную, большую по объему программу, но и сделает ее к тому же нечитабельной и неудобной в сопровождении. Лучший способ состоит в том, чтобы сначала создать хорошо структурированную программу, а затем в ходе эксплуатационных испытаний найти то, что делает ее медленной или большой. Этот способ при программировании дает возможность сосредоточиться на особенностях C++ и таким образом построить с большой вероятностью программу, которая менее всего нуждается в кардинальной оптимизации.

Имейте в виду, что в большинстве случаев оптимизация достигается путем компромиссов. Вы можете увеличивать быстродействие программы (скорость ее выполнения), но тогда, вероятно, придется сделать ее больше, или же использовать больший объем памяти для ее выполнения.

Встроенный код

В С++ предусмотрен простой способ увеличения быстродействия ценой увеличения объема памяти.

Сложная объектно-ориентированная программа может делать миллионы вызовов функций в секунду. Даже если каждое обращение к функции занимает очень мало времени, сокращение накладных расходов на такие вызовы может быть иногда настолько существенным, что позволяет превратить недопустимо медленную программу во вполне приемлемую.

Если испытание указывает, что уменьшение количества вызовов функций может повысить эффективность (причем для существенной оптимизации обычно нужно устранить только незначительное (в коде) количество вызовов, которые повторяются достаточно часто), то можно устранить накладные расходы на обращения к функции за счет увеличения размера программы. Чтобы устранить накладные расходы на обращения к функции, код функции можно *встроить* на месте ее вызова.

Есть два способа встраивания кода. Самый простой состоит в том, чтобы поместить реализацию функции в объявление класса. Это сделано в приведенной ниже (нешаблонной) версии `anAccumulator`:

[illegible]

```

*12:         return myValue;
*13:     }
14:
15:     virtual anAccumulator
15:    &ReferenceToANewAccumulator(void) = 0;
16:
17:     protected:
18:
19:     float myValue;
20: }

```

Анализ

В строках 10–13 приведена встроенная функция Value(). (Конечно, она представляет собой механизм доступа, т.е. является получателем.) Механизмы доступа (получатели) и механизмы установки (модификаторы) очень часто являются вполне подходящими кандидатами на встраивание, потому что они маленькие, а вызываются часто.

В результате этого изменения, всюду, где в программе осуществляется доступ к значению myValue (т.е. вызывается Value()), будет вставлен код, который получает содержимое myValue, но не будет сгенерирован код, который находит место в стеке, вызывает функцию, получает результат из стека и выталкивает его из стека. Но помните, что современные компьютеры и компиляторы разработаны так, что выполняют обращения к функции и управление стеком очень быстро, так что этот прием не может значительно повысить эффективность.

Вы можете встраивать виртуальные функции так же, как и функции, не являющиеся виртуальными, но компилятор может проигнорировать ваш запрос и создать обычное обращение к функции. Виртуальные функции компилятор встраивает только в том случае, если он абсолютно уверен, что знает фактический класс, для экземпляра которого вызывается функция.

Но даже функции, которые не являются виртуальными, могут быть встроены или вызваны обычным способом по усмотрению компилятора. Например, функция, которая вызывает себя, встроена не будет.

Член-функцию можно также встроить в файл реализации, чтобы не открывать ее реализации. Для этого нужно воспользоваться ключевым словом inline (встроить) в заголовке функции (в файле реализации) следующим образом:

```
1: inline float anAccumulator: :Value(void) const
2: {
3:     return myValue;
4: }
```

Но помните также, что компилятор в первую очередь стремится сделать корректную программу и в некоторых случаях проигнорирует ключевое слово `inline` (встроить), если ему покажется, что из-за встраивания могут возникнуть проблемы.

Приращение (увеличение) и уменьшение

Почти каждый современный процессор может увеличивать или уменьшать значение переменной одной машинной командой (и многие процессоры могут выполнять эти действия даже в какой-либо части команды). Операторы `++` и `--` позволяют компиляторам C++ генерировать специальные команды приращения и уменьшения (декремента) в объектном коде, который является результатом трансляции исходной программы, написанной на C++. Так, используя

```
Index++;
```

а не

```
Index = Index + 1;
```

вы поможете компилятору ускорить выполнение программы.

Не исключено также, что компилятор сможет оптимизировать

```
Index+= 3;
```

лучше, чем

```
Index = Index + 3;
```

и потому по тем же причинам первая инструкция будет выполняться быстрее, чем вторая.

Обратите внимание, что это, однако, не относится к перегруженным операторам, потому что они имеют определяемые пользователем реализации.

Шаблоны или универсальные классы?

Шаблоны генерируют код в вашей программе каждый раз, когда по ним создается экземпляр, так что они увеличивают программу в большей степени, чем универсальные классы. Однако в универсальных классах (которые усиленно используют полиморфизм класса) приходится использовать виртуальные функции, а каждое обращение к такой функции выполняется чуть-чуть медленнее, чем к обычной.

В любом случае, не эффективность должна определять ваш выбор в пользу шаблонов или полиморфизма класса, если, конечно, проблемы не настолько серьезны, что с помощью измерения эффективности вы установили, что они являются результатом использования шаблонов (или результатом отказа от использования шаблонов).

Хронометраж кода

Чтобы выяснить, какие части кода выполняются медленно, нужно обязательно выполнить хронометраж кода. Это — превосходное приложение инструментальных средств тестирования.

В Стандартной библиотеке для C есть функция `time`, которая возвращает время. Ее можно использовать и для хронометража кода. К сожалению, приращения времени она измеряет только с точностью до одной-двух секунд, так что приходится выполнять много испытаний, чтобы получить время, которое можно измерить с ее помощью. Например:

```
1: #include <time.h>
2:
3: time_t Start; // Начало
4: time_t End; // Конец
5:
6: time(&Start);
7:
8: for (int Index = 0; Index < 100000; Index++)
9: { // Значение = Сумматор.Значение
10:     int Value = Accumulator.Value();
11: };
12:
```

```

13: time(&end);
14:           // Время от начала до конца
15: double TimeRequired = difftime(Start,End);

```

Имейте в виду, что некоторые операции не могут быть легко или надежно выполнены в таких циклах. Например:

```

1: #include <time.h>
2:
3: time_t Start; // Начало
4: time_t End;   // Конец
5:
6: time(&Start);
7:
8: for (int Index = 0; Index < 100000; Index++)
9: { // Значение = Сумматор.Применить
10:     int Value =
        Accumulator.Apply(aRequest(aRequest::add,34));
11: };
12:
13: time(&end);
14:           // Время от начала до конца
15: double TimeRequired = difftime(Start,End);

```

Этот код выдает 100 000 запросов на сумматор, а это означает, что фактически проверяется эффект увеличения ленты, а не быстродействие вычислителя. Всегда нужно удостовериться, что вы правильно интерпретируете результаты измерений.

В любом случае, только должным образом собранные результаты хронометража позволяют точно определить, какие части программы нуждаются в оптимизации.

Чтобы выяснить, действительно ли изменения помогли ускорить выполнение программы, можно встроить хронометраж в инструментальные средства тестирования.

Кроме того, чтобы определить, на что ваша программа тратит основное время, можно использовать профилировщик, о чем уже говорилось в материалах урока 23, "Испытание объектов с помощью наследования".

Размер программы и структуры данных

Размер программы может влиять на время, которое требуется для запуска. Кроме того, большая программа или программа с большими структурами данных может занимать так

много памяти, что операционная система будет вынуждена постоянно перемещать части программы из оперативной памяти на диск и обратно, а это существенно снижает эффективность.

К сожалению, управлять размером может быть очень трудно. Чтобы выполнить все запрошенные пользователями функции и обеспечить графические интерфейсы пользователя, необходимые в большинстве программ, наиболее современные объектно-ориентированные программы используют обширный и сложный набор внутренних библиотек и библиотек независимых производителей. Обойтись без этих библиотек можно только в очень редких случаях и почти никогда нельзя уменьшить их размер. Лучшее, на что можно надеяться, — что флажок оптимизации компилятора заставит компилятор пропустить весь тот код, который никогда не вызывается.

Чтобы повысить быстродействие, часто структуры данных располагаются в оперативной памяти. Иногда можно уменьшить их размер, удаляя их из динамически распределяемой области памяти сразу после того, как необходимость в них отпала, или же размещая их в стеке, а не в динамически распределяемой области памяти.

Иногда через определенные промежутки времени полезно измерять объем памяти, занимаемый программой. Ищите большие объекты, которые висят в памяти почти от начала выполнения программы до ее завершения, несмотря на то, что нужны они очень редко.

Резюме

Оптимизировать быстродействие можно только в том случае, если начать с хорошо структурированной и удобной в сопровождении объектно-ориентированной программы. Приступая к оптимизации такой программы, начать нужно с тщательного хронометража, чтобы удостовериться, что вы на самом деле нашли причину снижения эффективности. Найдя причину, попробуйте применить возможности языка C++, которые помогут повысить эффективность.

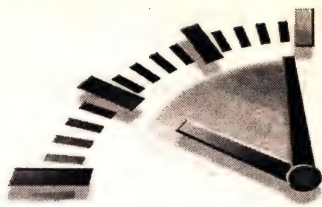
Как только вы внесете намеченные изменения, повторно выполните все необходимые тесты, чтобы определить, достигли ли вы нужного эффекта.

Однако имейте в виду, что многие проблемы с эффективностью являются следствием проблем в логике вашей программы или обусловлены накладными расходами на доступ к файлам или базам данных. Уже написано немало учебников по алгоритмам, в которых обсуждается изменение быстродействия в результате применения альтернативных подходов к решению задач программирования, и необходимо заглянуть в эти учебники, чтобы оценить влияние альтернативных подходов на оптимизацию программы.

Наконец, не забывайте, что размер структур данных и размер сгенерированного кода программы может влиять на эффективность. Чтобы уменьшить программу, вероятно, в какой-то степени придется пожертвовать быстродействием.

УРОК 27

Итоги, или повторение пройденного



В этом уроке, наконец, мы подведем итоги изучения представленного материала и обсудим способы усовершенствования калькулятора.

Как усовершенствовать калькулятор?

Читая эту книгу, вы с самого ее начала работали над калькулятором, преобразовывая его снова и снова, добавляя к нему все новые и новые возможности, реорганизуя его или же просто упрощая его для понимания.

Теперь вы можете самостоятельно продолжать эту работу и добавить к калькулятору некоторые возможности.

Добавление возможностей отмены ввода (Undo) и повторения ввода (Redo)

Вы можете отменить результаты вычислений, обнуляя сумматор и повторно считывая ленту до запроса, который предшествует последнему записанному на ленте запросу. Вы можете даже добавить две дополнительные ленты — ленту отмен, которая содержит все запросы, которые не были отменены, и ленту восстановления, которая содержит все отмененные запросы. Тогда оператор отмены может стереть последний элемент на ленте отмен и записать его на ленту восстановления, обнулить сумматор и повторно прочесть ленту отмен.

Добавление поименованных переменных

Стандартная библиотека C++ предлагает контейнер, называемый `map` (отображение, соответствие, карта), который может использоваться для хранения поименованных значений. Калькулятор, который позволяет вводить значение вместе с его названием и затем использовать это название (имя) в вычислении, намного полезнее разработанного вами калькулятора.

Использование калькулятора в более мощной программе, например в электронной таблице

Вы можете также создать ориентированную на текстовое представление данных программу, подобную электронной таблице, — калькулятор будет в ней генерировать результаты для каждой ячейки этой электронной таблицы. На самом деле, можно даже создать отдельный экземпляр калькулятора для каждой ячейки электронной таблицы — это значительно упростило бы программу. Вы можете применять статические переменные в классах — тогда ячейки смогут совместно использовать информацию.

Использование графического интерфейса пользователя (Graphical User Interface, GUI)

Вы можете добавить оконный интерфейс пользователя к вашему калькулятору или электронной таблице. Созданные вами объекты в значительной степени являются независимыми от внешнего интерфейса, а это означает, что очень просто заменить символично-ориентированный интерфейс пользователя на интерфейс для Windows, Unix или Macintosh.

Изученные уроки

Вы начали с программы всего в несколько коротких строк и довели ее до полностью объектно-ориентированного набора из нескольких классов, записанных во многих файлах.

Если вы профессиональный программист, вы должны теперь понимать, насколько хорошо C++ поддерживает каждый этап разработки программ.

А если вы никогда не программировали прежде, вы имели возможность пройти несколько полных циклов разработки и сопровождения, принимая участие во всем, что делает профессиональный программист.

Думайте о классах и объектах

Вы можете рассматривать программу как список команд, набор функций или набор классов и объектов. Но наиболее ценным является взгляд на программу как на совокупность классов и объектов, поскольку именно такой взгляд открывает самые большие возможности для того, чтобы наиболее полно отразить результат размышлений о системе в структуре программы и всей системы. А потому побольше думайте о классах и объектах, а не о командах и функциях.

Классы и объекты поддерживаются следующими средствами C++:

- объявление класса и его определение;
- создание экземпляра в стеке;
- создание экземпляра и его удаление в динамической памяти;
- виртуальные функции и полиморфизм класса;
- перегрузка функций и операторов;
- шаблоны.

Развитие программы

В ходе создания этой программы вы видели, что C++ поддерживает и процедурное, и объектно-ориентированное программирование, и что в этом языке есть средства, помогающие программисту плавно, без скачков преобразовать простую программу в сложную.

C++ — превосходный язык для развития программ с помощью применения многих видов и уровней абстракции, поддерживаемых в C++. Перечислим кратко изученные вами виды и уровни абстракции:

- абстракция функций;
- абстракция данных;
- модули;
- классы и объекты с член-функциями и переменными;
- наследование и полиморфизм класса;
- шаблоны.

Частое улучшение кода (переразложение на классы)

Всегда помните, что улучшение кода (переразложение на классы) программы в новые конфигурации этих абстракций накладывает критическую ответственность на программиста. Осторожно улучшая код (переразлагая программу на классы) по мере увеличения сложности, можно сделать программу более понятной и простой. Если же код не переразлагать на классы, постоянно увеличивающиеся запросы службы сопровождения постепенно приведут программу в хаотическое состояние.

Поддержка контракта

Объектно-ориентированное программирование работает потому, что объекты представляют собой интерфейсы, которые заключает контракты с любыми пользователями объекта. Вы улучшили код и много раз заново реализовывали эту программу и ее объекты, и каждый раз убеждались, что поведение объектов и функций было именно таким, какое от них ожидалось.

C++ предлагает много средств, которые помогают поддерживать контракт:

- строгий контроль типов, осуществляемый компилятором;
- требование объявления типов, констант и переменных до их использования;
- наследование и полиморфизм класса;
- абстрактные классы (интерфейсы);
- множественное наследование;
- шаблоны.

Частые испытания

После любого изменения программу нужно протестировать, а вы сделали так много изменений! И потому пришлось выполнить так много испытаний. Теперь вы знаете, что C++ облегчает создание многих видов инструментальных средств тестирования, и умеете проверить каждый уровень абстракции. Объекты могут проверять себя или друг друга, их можно даже объединить в подсистемы в целях применения инкрементального тестирования (испытания добавлений или изменений).

Подумайте об эффективности в подходящее время

Надеюсь, эта книга вдохновила вас на размышления о том, как лучше организовать программы с помощью средств языка C++. Если программа хорошо структурирована, гораздо проще повысить ее эффективность, оптимизируя конкретные объекты с учетом результатов проведенных испытаний и хронометража.

Не усложняйте простых вещей...

C++ — тщательно разработанный язык с широким разнообразием возможностей, многие из которых вы уже изучили, но некоторые из них были изучены лишь поверхностно, а некоторые ради простоты были опущены совсем. Помните, что для достижения максимальной понятности каждую возможность языка нужно использовать наиболее подходящим способом.

Чтобы упростить сам калькулятор и чтение его программы, вы непрерывно улучшали код и убрали все “ветхие” конструкции в нем. И эти усилия всегда вознаграждались снижением риска внесения новых ошибок в ходе сопровождения программы.

Соблюдайте соглашения об именовании

Работа программы не зависит от того, какими именами вы называете в ней те или иные вещи. А называть вещи можно по-разному: можно давать им понятные названия, а можно — какие-нибудь несусветные обозначения. Но правильное обозначение вещей и тщательный подбор имен облегчают обна-

ружение того, что вы делаете неправильно, и упрощают понимание незнакомых разделов программы.

В этой книге используется соглашение об именовании, в соответствии с которым дополнительно к (смысловому) значению идентификатора в контексте программы указывается тип, источник, область видимости и время существования вещи, обозначаемой данным идентификатором. Это соглашение об именовании — только одно из многих, которые можно использовать на практике, и если вы используете библиотеки классов, разработанные другими программистами, вам придется изучить многие другие соглашения об именовании и адаптироваться к ним, притом, возможно даже придется использовать несколько соглашений об именовании одновременно. Хотелось бы надеяться, что вы всегда постараетесь уделять обозначениям такое же внимание, как и структуре и цели программы.

Будьте терпеливы, работая с компилятором

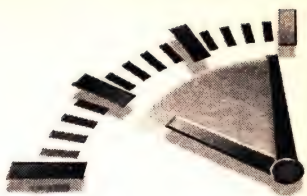
Теперь вы уже немного знаете о том, как “думает” компилятор. Это помогает, когда вы получаете запутывающее сообщение об ошибках или когда кое-что, что кажется совершенно правильным, никак не хочет компилироваться. Будьте терпеливы — это поможет вам, когда программа не будет работать вообще или будет непрерывно выдавать ошибочные результаты. И вы освоите все методы, которые нужны для разработки высоконадежных программы с очень малым количеством ошибок.

Ни одна программа не совершенна, так что любая программа может быть улучшена. Поскольку вы хорошо разобрались в языке C++ и в программировании, вы можете изменить и адаптировать ваши программы, применяя новые знания.

Будьте терпеливы — ведь все, что требуется, — это время, опыт и размышления.

Приложение А

Операторы



В этом приложении в таблице перечислены и описаны многие операторы языка C++. Многие из этих операторов можно перегрузить в разработанных вами классах.

При перегрузке операторов вы не ограничены типами, указанными в таблице. Часто типом будет ваш класс. Вы можете перегрузить любой инфиксный оператор несколько раз и работать с различными типами, но не забывайте, что возвращаемый тип не является частью сигнатуры и потому он не принимается во внимание при выборе перегруженной версии.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Арифметические операторы</i>					
+	Добавить	Число или символ (char)	Да	a+b	Сложить a и b
-	Вычесть	Число или символ (char)	Да	a-b	Вычесть b из a
*	Умножить	Число или символ (char)	Да	a*b	Умножить a и b
/	Делить	Число или символ (char)	Да	a/b	Разделить a на b
++	Приращение, увеличение, инкремент	Число или символ (char)	Да	a++	Выдать a как результат, а затем увеличить a на 1
++	Приращение, увеличение, инкремент	Число или символ (char)	Да	++a	Увеличить a на 1 и выдать увеличенное значение как результат
--	Уменьшение, декремент	Число или символ (char)	Да	a--	Выдать a как результат, а затем уменьшить a на 1

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
Арифметические операторы					
--	Уменьшение, декремент	Число или символ (char)	Да	--a	Уменьшить a на 1 и выдать уменьшенное значение как результат
%	Модуль, остаток от деления	Число или символ (char)	Да	a%b	Найти остаток от деления a на b
Операторы присваивания					
=	Присваивание	Любой	Да	a=b	Сделать содержимое a таким же, как и содержимое b
+=	Арифметический оператор присваивания +=	Число или символ (char)	Да	a+=b	Эквивалент a=a+b
-=	Арифметический оператор присваивания -=	Число или символ (char)	Да	a-=b	Эквивалент a=a-b
*=	Арифметический оператор присваивания *=	Число или символ (char)	Да	a*=b	Эквивалент a=a*b
/=	Арифметический оператор присваивания /=	Число или символ (char)	Да	a/=b	Эквивалент a=a/b

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Операторы сравнения и логические</i>					
!	Отрицание	Логический (bool)	Да	!a	Если a имеет значение true (истина), присвоить ему значение false (ложь); если a имеет значение false (ложь), присвоить ему значение true (истина)
==	Равно	Любой	Да	a == b	Если содержимое a равно содержимому b, это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)
!=	Не равно	Любой	Да	a != b	Если содержимое a не равно содержимому b, это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Операторы сравнения и логические</i>					
>	Больше	Любой	Да	$a > b$	Если содержимое a больше содержимого b , это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)
<	Меньше	Любой	Да	$a < b$	Если содержимое a меньше содержимого b , это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)
>=	Больше или равно	Любой	Да	$a >= b$	Если содержимое a больше или равно содержимому b , это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Операторы сравнения и логические</i>					
<=	Меньше или равно	Любой	Да	a <= b	Если содержимое a меньше или равно содержимому b, это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)
&&	И	Логический (bool)	Да	a && b	Если a и b имеют значение true (истина), это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)
	Или	Логический (bool)	Да	a b	Если a или b имеет значение true (истина), это выражение принимает значение true (истина); в противном случае это выражение принимает значение false (ложь)

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Поразрядные операторы¹</i>					
&	И	Целое число, символ (char) или логиче- ский (bool)	Да	a & b	Результатом является число, в котором уста- новлены в 1 те разря- ды, в которых они рав- ны 1 в a и в b
	Или	Целое число, символ (char) или логиче- ский (bool)	Да	a b	Результатом является число, в котором уста- новлены в 1 те разря- ды, которые равны 1 в a или в b
^	Исключительное или	Целое число, символ (char) или логиче- ский (bool)	Да	a ^ b	Результатом является число, в котором уста- новлены в 1 те разряды, которые имеют разные значения в a и b
<<	Сдвиг влево	Целое число, символ (char) или логиче- ский (bool)	Да	a << b	Результатом является число, в котором раз- ряды a сдвинуты на b позиций влево

¹ Поразрядные операторы выполняются над отдельными битами простого типа; обычно в символе 8 битов, 16 битов в short int (короткий int), 32 бита в int и т.д. Поразрядные операторы выполняются так, как если бы их операнды были двоичными числами, независимо от их фактического типа.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
Поразрядные операторы²					
>>	Сдвиг вправо	Целое число, символ (char) или логический (bool)	Да	a >> b	Результатом является число, в котором разряды a сдвинуты на b позиций вправо
	Инвертирование (иногда дополнение) или поразрядное отрицание	Целое число, символ (char) или логический (bool)	Да	~ a	Результатом является число, в котором разряды a инвертированы. Это значит, что разряды результата равны 1, если соответствующие им разряды в a равны 0, и равны 0, если соответствующие им разряды в a равны 1
&=	Побитовый (поразрядный) оператор присваивания И	Целое число, символ (char) или логический (bool)	Да	a &= b	Результатом является число, разряды которого получаются после выполнения операции И над соответствующими разрядами a и b

² Поразрядные операторы выполняются над отдельными битами простого типа; обычно в символе 8 битов, 16 битов в short int (короткий int), 32 бита в int и т.д. Поразрядные операторы выполняются так, как если бы их операнды были двоичными числами, независимо от их фактического типа.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Поразрядные операторы³</i>					
=	Побитовый (поразрядный) оператор присваивания Или	Целое число, символ (char) или логический (bool)	Да	a = b	Результатом является число, разряды которого получаются после выполнения операции Или над соответствующими разрядами a и b
^=	Побитовый (поразрядный) оператор присваивания Исключительное или	Целое число, символ (char) или логический (bool)	Да	a ^= b	Результатом является число, разряды которого получаются после выполнения операции Исключительного или над соответствующими разрядами a и b
>>=	Побитовый (поразрядный) оператор присваивания со сдвигом вправо	Целое число, символ (char) или логический (bool)	Да	a >>= b	Результатом является число, в котором разряды a сдвинуты на b позиций вправо

³ Поразрядные операторы выполняются над отдельными битами простого типа; обычно в символе 8 битов, 16 битов в short int (короткий int), 32 бита в int и т.д. Поразрядные операторы выполняются так, как если бы их операнды были двоичными числами, независимо от их фактического типа.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Поразрядные операторы⁴</i>					
<<=	Побитовый (поразрядный) оператор присваивания со сдвигом влево	Целое число, символ (char) или логический (bool)	Да	a >>= b	Результатом является число, в котором разряды a сдвинуты на b позиций влево
<i>Операторы, связанные с указателями⁵</i>					
&	Адрес	Любой	Да (но не рекомендуется)	&a	Результатом является адрес a
*	Разыменованное	Любой	Да (но не рекомендуется)	*a	Результатом является содержимое, хранящееся по адресу a
++	Увеличение, обращение, инкремент	Указатель	Да	a++	Результатом является a, а затем a увеличивается так, чтобы он указывал в памяти на следующий элемент того же типа, на который указывает a

⁴ Поразрядные операторы выполняются над отдельными битами простого типа; обычно в символе 8 битов, 16 битов в short int (короткий int), 32 бита в int и т.д. Поразрядные операторы выполняются так, как если бы их операнды были двоичными числами, независимо от их фактического типа.

⁵ Все арифметические операции могут быть выполнены над указателями. Компилятор не проверяет, действительно ли указатель указывает на подходящий объект в памяти.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Операторы, связанные с указателями⁶</i>					
++	Увеличение, приращение, инкремент	Указатель	Да	++a	Результатом является a, увеличенный так, чтобы он указывал в памяти на следующий элемент того же типа, на который указывает a
--	Уменьшение, декремент	Указатель	Да	a--	Результатом является a, а затем a уменьшается так, чтобы он указывал в памяти на предыдущий элемент того же типа, на который указывает a
--	Уменьшение, декремент	Указатель	Да	--a	Результатом является a, уменьшенный так, чтобы он указывал в памяти на предыдущий элемент того же типа, на который указывает a

⁶ Все арифметические операции могут быть выполнены над указателями. Компилятор не проверяет, действительно ли указатель указывает на подходящий объект в памяти.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
<i>Оператор ссылки</i>					
&	Адрес	Любой	Да (но не рекомендуется)	&a	Результатом является адрес, на который ссылается a
<i>Операторы приведения⁷</i>					
static_cast	Статическое приведение	Любой	Нет (но можно использовать перегрузку традиционного приведения)	static_cast<тип>(a)	Приводит a к типу, если это допускается правилами приведения типов или если перегружено традиционное приведение
cast	(Традиционное) приведение	Любой	Да	<тип>(a)	Приводит a к типу независимо от того, допускается ли это правилами приведения типов
dynamic_cast	Динамическое приведение или приведение класса	Любой указатель на класс или ссылка	Нет	dynamic_cast<другой_класс*>(&a) или dynamic_cast<другой_класс*>(a)	Приводит указатель на данный класс или ссылку к указателю или ссылке на указанный суперкласс или производный класс

⁷ Операторы приведения конвертируют (преобразовывают) один тип к другому. Более подробная информация приведена разработчиком C++ на сайте <http://anubis.dkuug.dk/JTC1/SC22/WG21/docs/papers/1993/N0349a.pdf>.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
Операторы приведения⁸					
<code>const_cast</code>	Приведение с помощью <code>const</code> (константа)	Любая переменная	Нет	<code>const_cast<тип>(a)</code>	Рассматривает данный класс или указатель как <code>const</code> (константа) или не как <code>const</code> (константа) в зависимости от приведения
<code>reinterpret_cast</code>	Приведение с новой интерпретацией	Любая переменная	Нет	<code>reinterpret_cast<тип>(a)</code>	Рассматривает данную переменную как имеющую <code>тип</code> , независимо от того, допускает ли система типов такое приведение; это совпадает с традиционным приведением
Условный оператор					
<code>?:</code>	Условное выражение	Любой	Нет	<code>a ? b : c</code>	Если <code>a</code> представляет логическое значение <code>true</code> (истина), то в качестве результата возвращается значение <code>b</code> , в противном случае — значение <code>c</code>

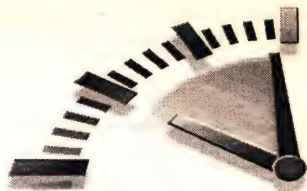
⁸ Операторы приведения конвертируют (преобразовывают) один тип к другому. Более подробная информация приведена разработчиком C++ на сайте <http://anubis.dkuug.dk/JTC1/SC22/WG21/docs/papers/1993/N0349a.pdf>.

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
Операторы распределения памяти					
<code>new</code>	Оператор <code>new</code> (новый, создать)	Тип	Да	<code>new тип</code>	Указатель на область памяти, достаточно большую для хранения данного, имеющего тип <code>тип</code>
<code>delete</code>	Удалить	Любой указатель	Да	<code>delete a</code>	Возвращает память, занятую переменной <code>a</code> , обратно в динамическую область памяти
<code>delete</code>	Удалить	Любой указатель	Да	<code>delete [] a</code>	Возвращает память, занятую массивом <code>a</code> , обратно в динамическую область памяти
Оператор области видимости					
<code>::</code>	Оператор разрешения области видимости	Имена	Нет	<code>имя_пространства_имен::имя</code> или <code>имя_пространства_имен::класс::имя</code> или <code>класс::вложенный_класс::имя</code>	Разрешает конфликт имен, указывая контекст (пространство имен или класс), в котором данное имя может быть найдено компилятором
Операторы доступа к члену					
<code>.</code>	Оператор выбора точки или член	Структурный тип или тип класса и член	Нет	<code>объект.член</code>	Результатом является содержимое члена или вызов члена

Оператор	Названия и значения	Тип операнда	Возможность перегрузки	Пример	Пояснение
Операторы доступа к члену					
->	Оператор стрелка или оператор выбора члена по указателю	Структурный тип или тип класса и член	Нет	<i>указатель_на_объект</i> <i>->имя_члена</i>	Разыменовывает указатель на объект и выдает содержимое члена или вызов члена
Операторы указания на член					
. *	Указатель на член	Экземпляр класса (объект) и указатель на член	Нет	<i>a.*b</i>	Получают содержимое того члена класса <i>a</i> , на который указывает <i>b</i>
-> *	Указатель на член	Указатель на класс и указатель на член класса	Нет	<i>a->*b</i>	Получают содержимое того члена класса <i>*a</i> , на который указывает <i>b</i>
Операторы вызова функций и индексирования					
()	Оператор обращения к функции	Имя функции или объект с перегрузкой	Да	<i>имя()</i>	Вызывает функцию, которая может быть перегружена в классе
[]	Оператор индексирования	Имя	Да	<i>имя[]</i>	Получает элемент массива или вызывает перегруженную в классе функцию

ПРИЛОЖЕНИЕ Б

Старшинство операторов



Важно знать, что операторы имеют старшинство, но не так уж важно запомнить точное значение старшинства (или ранга).

Старшинство операторов определяет порядок, в котором программа выполняет операции в формуле. Сначала выполняются старшие операторы.

Старшие операторы “привязывают” к себе операнды сильнее, чем операторы с меньшим старшинством; именно поэтому сначала вычисляются старшие операторы. В табл. Б.1 операторы C++ приведены в порядке их старшинства.

Таблица Б.1. Старшинство операторов

<i>Ранг</i>	<i>Название</i>	<i>Оператор</i>
1	разрешение области видимости	::
2	выбор члена	. ->
	индексирование	[]
	обращение к функции	()
	постфиксное приращение	++
	декремент	--
3	префиксное увеличение	++
	префиксный декремент	--
	дополнение (инверсия)	~
	и	&
	не	!
	одноместный минус	-
	одноместный плюс	+
	адрес	&

Окончание табл. Б.1

Ранг	Название	Оператор
	разыменование	*
	создать (новый)	new
	создать (новый) массив	new[]
	удалить	delete
	удалить массив	delete[]
	приведение	()
	размер	sizeof()
4	выбор члена для указателя	.* ->*
5	умножение	*
	деление	/
	остаток от деления (модуль)	%
6	сложение	+
	вычитание	-
7	сдвиг влево	<<
	сдвиг вправо	>>
8	отношения неравенства	< <= > >=
9	равно	==
	не равно	!=
10	поразрядное и	&
11	поразрядное исключитель- ное или	^
12	поразрядное или	
13	логическое и	&&
14	логическое или	
15	условный оператор	?:
16	присваивание	= *= /= %= += -= <<= >>= &= = *=
17	вызов исключения	throw
18	запятая	,

Предметный указатель

!

!, оператор, 324; 337
!=, оператор, 324; 338

#

#define, команда, 146
#endif, команда, 146
#ifdef, команда, 146

%

%, оператор, 323; 338
%=, оператор, 338

&

&&, оператор, 326; 338
&, оператор, 327; 330; 332;
337; 338; 337; 338
&=, оператор, 328; 338

(

(), оператор, 335; 337

*

*, оператор, 322; 330; 338
*=, оператор, 323; 338

,

-, оператор, 322
,, оператор, 338

.

.*, оператор, 335; 338
., оператор, 334; 337

/

/, оператор, 322; 338
/=: оператор, 323; 338

[

[], оператор, 335; 337

^

^, оператор, 327; 338
^=: оператор, 329

|

|, оператор, 327; 338
||, оператор, 326; 338
|=, оператор, 329; 338

~

~, оператор, 328; 337

+

+, оператор, 322; 337; 338;
337; 338
++, оператор, 118; 322; 330;
331; 337
+=", оператор, 323; 338

<

<, оператор, 325; 338
<<, оператор, 327; 338
<<=, оператор, 330; 338
<=, оператор, 326; 338

=

=, оператор, 323; 338
-=, оператор, 323; 338
==, оператор, 324; 338

>

->*, оператор, 335; 338
>, оператор, 325; 338
>, оператор, 335; 337
>=, оператор, 325; 338
>>, оператор, 328; 338
>>=, оператор, 329; 338

A

American National
Standards Institute, 26

B

bool, тип, 48
Boundary value tests, 265
break, инструкция, 107

C

Capacity tests, 263
cast, оператор, 332
char, тип, 48
const_cast, оператор, 333

D

default, инструкция, 106
delete, оператор, 334; 338
delete[], оператор, 338
DMOZ Open Directory
Project, 214
do, цикл, 90
double, тип, 48
dynamic_cast, оператор, 332

E

endl, манипулятор, 34

F

float, тип, 48
for, цикл, 116

G

gcc, 27

I

IDE, 27
ifstream, поток, 167
International Organization
for Standardization, 25
iostream, библиотека, 34
iostream.h, файл, 34
ISO, 25

L

long int, тип, 48

N

namespace, 83
new, оператор, 334; 338
new[], оператор, 338

O

ofstream, поток, 167
Out-of-bounds tests, 263

P

Performance tests, 263

R

Redo, 315
reinterpret_cast, оператор, 333

S

short int, тип, 47
sizeof(), оператор, 338
static_cast, оператор, 332
Stress tests, 263
switch, инструкция, 106

T

throw, оператор, 338

U

UML, 189
Undo, 315

Unified Modeling
Language, 189
unsigned long int, тип, 47
unsigned short int, тип, 47
using namespace, 86

W

while, цикл, 94
Within-bounds tests, 264

A

Абстрактный класс,
270; 271
Агрегация, 279
Адрес, 159
Американский националь-
ный институт стандар-
тов, 25
Анализ, 28
Аргумент, 68
Атрибут, 190

Б

Библиотека, 26; 214
iostream, 34
Блок, 30

В

Вложенные круглые скоб-
ки, 41
Встраивание кода, 308
Выбор элемента через ука-
затель, 156

Г

Глобальные переменные, 77

Д

Декрементирование, оператор, 118

Дерево наследований, 252

Деструктор, 185

 виртуальный, 258

Дихотомия, 146

Доступ

 защищенный, 254

З

Зависимость, 190

Заглушка, 277

Заголовков, 30

Защищенный доступ, 254

И

Индекс, 113

Инициализатор, 183

Инициализация, 52

 статическая, 286

Инкапсуляция, 111; 178

Инкрементирование, оператор, 118

Инстанцирование, 178

Инструкция

 break, 107

 default, 106

 switch, 106

Инструментальные средства тестирования, 261

Интегрированная среда разработки, 27

Интерфейс, 270

Интерфейс пользователя, 136

Испытания, 28

 boundary value tests, 265

 capacity tests, 263

 empty tests, 262

 out-of-bounds tests, 263

 performance tests, 263

 stress tests, 263

 within-bounds tests, 264

автоматизированные, 262

в допустимых пределах, 264

в условиях недостатка ресурсов, 263

всесторонние, 261

запись в файлы входных и выходных данных, 265

использование наследования, 266

классов с помощью заранее подготовленных тестов, 262

неавтоматизированные, 262

производительности, 263

регрессивные, 265

с выходом за допустимые границы, 263

с граничными значениями, 265

с пустыми данными, 262

эффективности, 263

Исходный текст, 26

К

Класс

 vector, 214

 абстрактный, 270; 271

Н

вызовы функций в производном классе, 253
 заглушка, 277
 конкретный, 271
 кукла, 277
 полиморфизм, 250
 производный, 243
 симулятор, 278
 тренажер, 277
 Ключевое слово
 virtual, 255; 258
 Коллизия имен, 197
 Команда
 #define, 146
 #endif, 146
 #ifdef, 146
 Комментарий, 36
 Компилятор, 26
 Компиляция, 28
 Компоновка, 28
 Компоновщик, 79
 Константа, 46
 Конструктор, 182
 виртуальный, 258
 копии, 216
 Кукла, 277

М

Манипулятор
 endl, 34
 Массив, 113
 Международная организация по стандартизации, 25
 Метод деления пополам, 146
 Множественное наследование, 279
 Модуль, 31; 79

Наследование, 243
 изображение в UML, 244
 множественное, 279
 уровни, 251

О

Область видимости, 169
 Обработка исключений, 63
 Обратный вызов, 163; 211
 Общедоступный, 179
 Объект, 177
 ОО-образец, 277
 ОО-шаблон, 277
 Оператор
 !, 324; 337
 !=, 324; 338
 %, 323; 338
 % =, 338
 &, 327; 330; 332; 337; 338;
 337; 338
 &&, 326; 338
 & =, 328; 338
 (), 335; 337
 *, 322; 330; 338
 * =, 323; 338
 ,, 338
 ., 334; 337
 .*, 335; 338
 /, 322; 338
 /=, 323; 338
 [], 335; 337
 ^, 327; 338
 ^ =, 329
 |, 327; 338
 ||, 326; 338
 |=, 329; 338
 ~, 328; 337
 +, 322; 337; 338; 337; 338

++, 118; 322; 330; 331;
337
+=, 323; 338
<, 325; 338
<<, 327; 338
<<=, 330; 338
<=, 326; 338
-=, 323; 338
=, 323; 338
==, 324; 338
->, 156; 180; 335; 337
>, 325; 338
->*, 335; 338
>=, 325; 338
>>, 328; 338
>>=, 329; 338
cast, 332
const_cast, 333
delete, 334; 338
delete[], 338
dynamic_cast, 332
new, 334; 338
new[], 338
reinterpret_cast, 333
sizeof(), 338
static_cast, 332
throw, 338
адрес, 330; 332
больше, 325
больше или равно, 325
выбора члена по указате-
лю, 335
выбора элемента, 180
выбора элемента с помо-
щью указателя, 180
выбора элемента через
указатель, 156
вычесть, 322
декремент, 322; 323; 331
декрементирование, 118
делить, 322

динамическое приведе-
ние, 332
добавить, 322
запятая, 145
и, 326; 327
или, 326; 327
инвертирование, 328
индексирования, 335
инкремента, 116; 322;
330; 331
инкрементирование, 118
исключительное или, 327
меньше, 325
меньше или равно, 326
не, 59
не равно, 324
обращения к функ-
ции, 335
остаток от деления, 323
отрицание, 59; 324
поразрядное отрица-
ние, 328
постфиксный, 118
постфиксный прираще-
ния, 116
префиксный, 118
приведение, 332; 333
приведение класса, 332
приращение, 118; 322;
330; 331
присваивание, 323
присваивания со сдвигом
влево, 330
присваивания со сдвигом
вправо, 329
равно, 324
разрешения области ви-
димости, 86; 180; 334
разыменование,
разыменовывания,
126; 330

сдвиг влево, 327
 сдвиг вправо, 328
 создать, 334
 сравнения, 99
 статическое приведе-
 ние, 332
 стрелка, 335
 точка, 334
 традиционное
 приведение, 332
 увеличение, 118; 322;
 330; 331
 удалить, 334
 указатель на член, 335
 уменьшение, 118; 322;
 323; 331
 умножить, 322
 условное выражение, 333
 Операция, 190
 Оптимизация
 операторы ++ и --, 310
 перегруженные операторы, 310
 размер программы и
 структуры данных, 312
 функция time, 311
 хронометраж, 311
 шаблоны и универсальные
 классы, 311
 Отладка, 28
 Отношение, 189

П

Параметр, 68
 необязательный, 140
 Параметрический тип, 290
 Перегрузка, 227
 <<, 234; 238
 ключевое слово const
 (константа), 238

ключевые положения, 239
 конструктора, 232
 конструктора копии, 240
 оператора, 233; 234; 236
 оператора <<, 234; 238
 операторы, 238
 присваиваний, 240
 Переменная
 на уровне класса, 284
 Переопределение
 члена суперкласса, 249
 Переразложение на клас-
 сы, 74
 Перечислимый тип, 151
 Подвыражение, 40
 Полиморфизм, 251
 указатели и ссылки, 251
 класса, 250
 Постоянная, 46
 Постфиксный оператор, 118
 Постфиксный оператор
 приращения, 116
 Поток
 ifstream, 167
 ofstream, 167
 Потомок, 243
 Предок, 243
 Препроцессор, 35
 Префиксный оператор, 118
 Приращение, оператор, 118
 Программирование отли-
 чий, 243
 Программирование с защи-
 той, 52
 Проектирование, 28
 Производный класс, 243
 Простая инструкция, 30
 Профилировщик, 264
 Публичный, 179

Р

Раздельная компиляция, 87
 Разрешение, 231
 Разыменовывание, 126
 Регрессивные испытания, 265
 Редактирование, 28
 Редактирование связей, 28
 Редактор, 26
 Рефакторинг, 74

С

Связный список, 157
 Сигнатура
 вызова, 228
 реализации, 228
 Симулятор, 278
 Сквозной контроль, 143
 Соглашения об
 именовании, 319
 Соединение частей, 279
 Создание экземпляра, 178
 Сокращение вычислений, 144
 Скрытие информации,
 111; 178
 Составная инструкция, 30
 Состояние программы, 167
 Специализация, 292
 Список
 связный, 157
 Средства тестирования, 261
 Ссылка, 132
 Стандартный C++, 25
 Статическая инициализация, 286
 Статическая функция, 285
 Структурный тип, 155
 Суперкласс, 243
 вызов, 259
 ссылка на реализацию, 259

Т

Тело, 30
 Тестирование, 28
 инструментальные средства, 261
 средства, 261
 Тильда, 185
 Тип
 bool, 48
 char, 48
 double, 48
 float, 48
 long int, 48
 short int, 47
 unsigned long int, 47
 unsigned short int, 47
 параметрический, 290
 перечислимый, 151
 структурный, 155
 Тренажер, 277

У

Увеличение, оператор, 118
 Указатель
 на функцию, 160
 Улучшение кода, 74
 Уменьшение, оператор, 118
 Унифицированный язык
 моделирования, 189
 Уровни наследования, 251
 Усечение, 53
 Условие, 59
 Условие цикла, 91
 Утечка памяти, 134

Ф

Файл
 iostream.h, 34

заголовочный, 79
 исполняемый, 87
 объектный, 87
 промежуточный, 87
 реализации, 79

Фигурные скобки, 30

Функция, 30; 67

main, 67

time, 311

абстрактная, 270

без аргументов и возвращаемых значений, 72

без аргументов, но с локальными переменными и возвращаемым значением, 72

встраивание кода, 308

встроенная виртуальная, 309

вызов, 74

главная, 67

класса, 285

общедоступная, 80

приватная, 80

с аргументами и возвращаемым значением, 73

статическая, 285

частная, 80

чистая виртуальная, 270

Ц

Цикл

do, 90

do-while, 94

for, 116

while, 94

инициализация, 116

с условием продолжения, 95

условие, 91; 117

шаг, 117

Ч

Чистая виртуальная функция, 270

Ш

Шаблон

в параметрах член-функций, 305

заголовочный файл, 291

и наследование, 305

инструкции #include, 291

контроль типов, 305

объявление, 290

определение, 290

реализация, 291; 305

риски, связанные с шаблонами, 305

специализация, 292

строгий контроль типов, 305

Шаблон проектирования

объектно-

ориентированный, 277

фабрика, 277

Э

Экземпляр, 178

Элемент, 113

Научно-популярное издание

Джесс Либерти

Освой самостоятельно C++

10 минут на урок, 2-е издание

Литературный редактор *С.Г. Татаренко*

Верстка *О.В. Мишутина*

Художественный редактор *В.Г. Павлютин*

Корректор *З.В. Александрова*

Издательский дом "Вильямс"

101509, г. Москва, ул. Лесная, д. 43, стр. 1

Изд. лиц. ЛР № 090230 от 23.06.99

Госкомитета РФ по печати

Подписано в печать 26.04.2004. Формат 84х108/32.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 11,0. Уч.-изд. л. 11,6.

Тираж 5000 экз. Заказ № **59.17.**

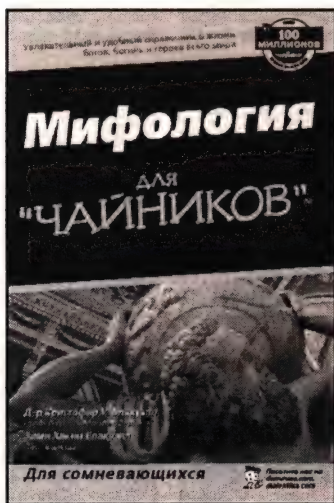
Отпечатано с готовых диапозитивов

в ООО "СЕВЕРО-ЗАПАДНЫЙ ПЕЧАТНЫЙ ДВОР"

188350, Ленинградская обл., г. Гатчина, ул. Солодухина, д. 2

МИФОЛОГИЯ ДЛЯ "ЧАЙНИКОВ"

**Кристофер У. Блакуэлл,
Эйми Хакни Блакуэлл**



www.dialektika.com

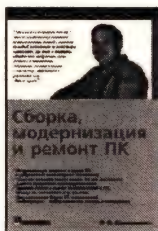
Мифология столь занимательна и поучительна, что человечество, несмотря на технический и технологический прогресс, так и не потеряло интереса к легендам давно прошедших времен. Древний Египет и классическая Греция, густонаселенная Индия и заснеженная Скандинавия, экзотические цивилизации майя и ацтеков далекой для европейцев Америки — мифы и сказания этих стран откроются перед читателями книги *Мифология для "чайников"*. Если вы еще верите в чудеса и еще не забыли наивную привязанность к сказкам, читайте эту книгу — вас ожидают встречи с легендарными богами и героями, чудовищами и красавицами, волшебниками и пройдохами.

в продаже

Мир книг для начинающих изучать ПК от издательской группы "ДИАЛЕКТИКА - ВИЛЬЯМС"



ISBN 5-8459-0322-X



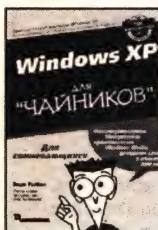
ISBN 5-8459-0478-1



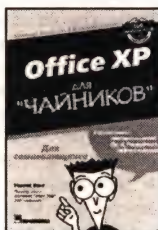
ISBN 5-8459-0383-1



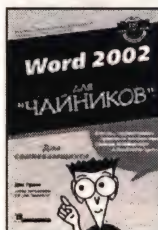
ISBN 5-8459-0325-4



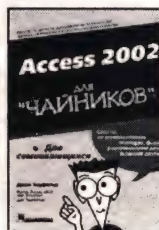
ISBN 5-8459-0266-5



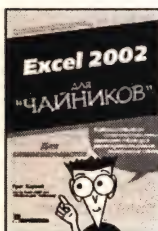
ISBN 5-8459-0279-7



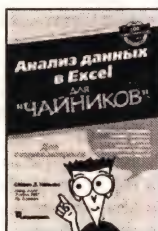
ISBN 5-8459-0267-3



ISBN 5-8459-0260-6



ISBN 5-8459-0227-4



ISBN 5-8459-0372-6



ISBN 5-8459-0377-7



ISBN 5-8459-0416-1

... и много других книг Вы найдете на наших сайтах



www.dialektika.com

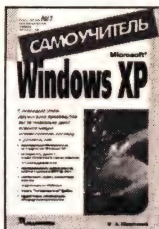


www.williamspublishing.com

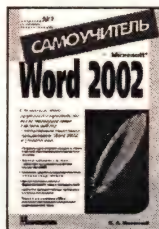
Серия книг "Самоучитель" от издательской группы "ДИАЛЕКТИКА-ВИЛЬЯМС"



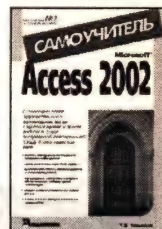
ISBN 5-8459-0383-1



ISBN 5-8459-0373-4



ISBN 5-8459-0415-3



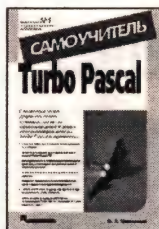
ISBN 5-8459-0468-4



ISBN 5-8459-0467-6



ISBN 5-8459-0466-8



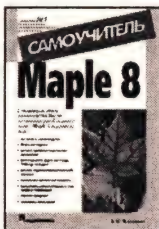
ISBN 5-8459-0477-3



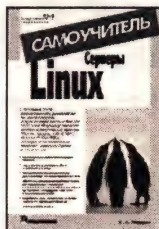
ISBN 5-8459-0427-7



ISBN 5-8459-0460-9



ISBN 5-8459-0452-8



ISBN 5-8459-0503-6



ISBN 5-8459-0502-8

... и много других книг Вы найдете на наших сайтах



www.dialektika.com



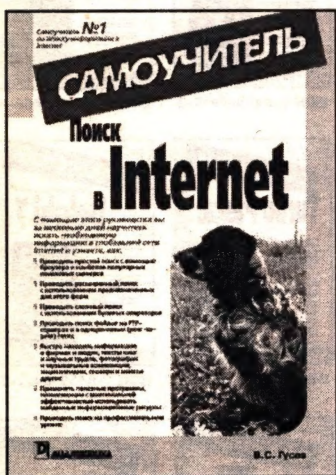
www.williamspublishing.com



www.ciscopress.ru

ПОИСК В INTERNET. САМОУЧИТЕЛЬ

Гусев В. С.



Самоучитель предназначен для тех, кто уже получил элементарные навыки работы в Internet и понимает: в Сети имеется огромное количество чрезвычайно полезной информации, но найти ее не так просто. В книге даны подробные рекомендации по проведению поиска разнообразных данных с помощью наиболее популярных поисковых машин, порталов, каталогов и т.д. Приведены подробные инструкции по выполнению сложных запросов на поиск и многочисленные примеры, благодаря которым даже неискушенный пользователь сможет быстро находить в Internet необходимую ему информацию.

www.dialektika.com

Плановая дата выхода
4 кв. 2003 г.

Интересные темы для программиста



ISBN 5-8459-0276-2



ISBN 5-8459-0250-9



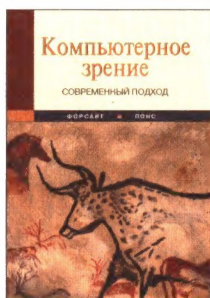
ISBN 5-8459-0437-4



ISBN 5-8459-0438-2



ISBN 5-8459-0527-3



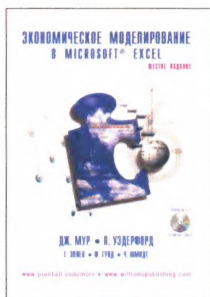
ISBN 5-8459-0542-7



ISBN 5-8459-0558-3



ISBN 5-8459-0406-4



ISBN 5-8459-0578-8

... и много других книг Вы найдете на наших сайтах



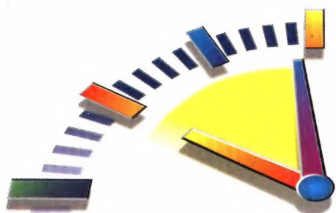
www.dialektika.com



www.williamspublishing.com



www.ciscopress.ru



SAMS
Освой самостоятельно!

C++

10 минут на урок ВТОРОЕ
ИЗДАНИЕ

Эта книга — простое практическое пособие для тех, кто хочет быстро получить результат. В материалах каждого из 27 уроков, рассчитанных на 10 минут, представлено важное для практического применения средство языка C++.



Советы

показывают
короткие
пути
к решениям



Предостережения

помогают
избежать
наиболее
распространен-
ных ошибок



Замечания

просто
и доходчиво
объясняют
новые термины
и определения

Нужно всего 10 минут, чтобы изучить:

- структуру программы
- переменные и константы
- классы и объекты
- функции и параметры
- перегрузку операторов
- инкапсуляцию
- наследование
- полиморфизм
- шаблоны
- исключения

ISBN 5-8459-0621-0

Категория: программирование

Содержание: ANSI C++

Уровень: для начинающих



SAMS

www.williamspublishing.com

www.sampublishing.com



0 4066



9 785845 906212

УЗРОВОЇ СЯМОСНОВАТВОЇ

С+С

10 МИНУТ НА УРОК

ВТОРОЕ ИЗДАНИЕ

Дж. Либерт

